ISSN 0265-2919

90p 74

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

READY MASTER

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

IN THE KNOW A look at a few of the ways information is represented in AI systems 1461

HARDWARE

THE TALKIES We sound out two devices that can give a home computer its own voice 1469

SOFTWARE

CONTROLLING THE CAST How to keep interactive characters under control 1466

THE ROCKFORD FILES Two cult games from the US that unearth hidden treasures 1480

COMPUTER SCIENCE

STACK 'EM UP We show how FORTH uses reverse Polish notation to simplify arithmetical operations 1475

JARGON

FROM RECURSION TO RELATION A weekly glossary of computing terms 1468

PROGRAMMING PROJECTS

ROLLING STONES Procedures which allow the computer to choose its own moves are designed this week 1472

MACHINE CODE

WHEN IN ROM A look at how it is possible to add your own BASIC commands to those of the Spectrum 1478

WORKSHOP

A BLOCK OFF THE OLD CHIP The seven components used in our multimeter 1464

Next Week

● Our artificial intelligence series concludes with an examination of future trends and developments.
● We design procedures that allow strategic moves to be played by the computer.
● Amstrad have recently repackaged their popular computer with a disk drive fitted in place of the cassette deck. We examine the potential of the new system.

QUIZ

1) What feature of a register allows faster access time than a normal memory location?
2) Why are BASIC array structures unsuitable for AI inference applications?
3) What is the difference between an allophone and a phoneme?
4) Why do the four and a half display digits in the multimeter project require only a single set of control lines?

Answers To Last Week's Quiz

1) The operating system on the BBC+ will 'steal' processor time by generating interrupts in the BASIC program, which switches to the 'shadow screen' display.
2) In FORTH, the character @ after a word will return its value, while ! will assign a value to the word.
3) A 'scout' is a transmission sent by a computer throughout the network which prepares the network to receive a transmission.
4) A logical record is part of a file that is defined according to software, whereas a physical record is a subdivision of a track that holds a fixed number of bytes.

Coming Up

- A look at GEM, Digital Research's 'operating environment' created for 16-bit machines such as the IBM PC.
- A series investigating ways in which computers have been employed in industry.

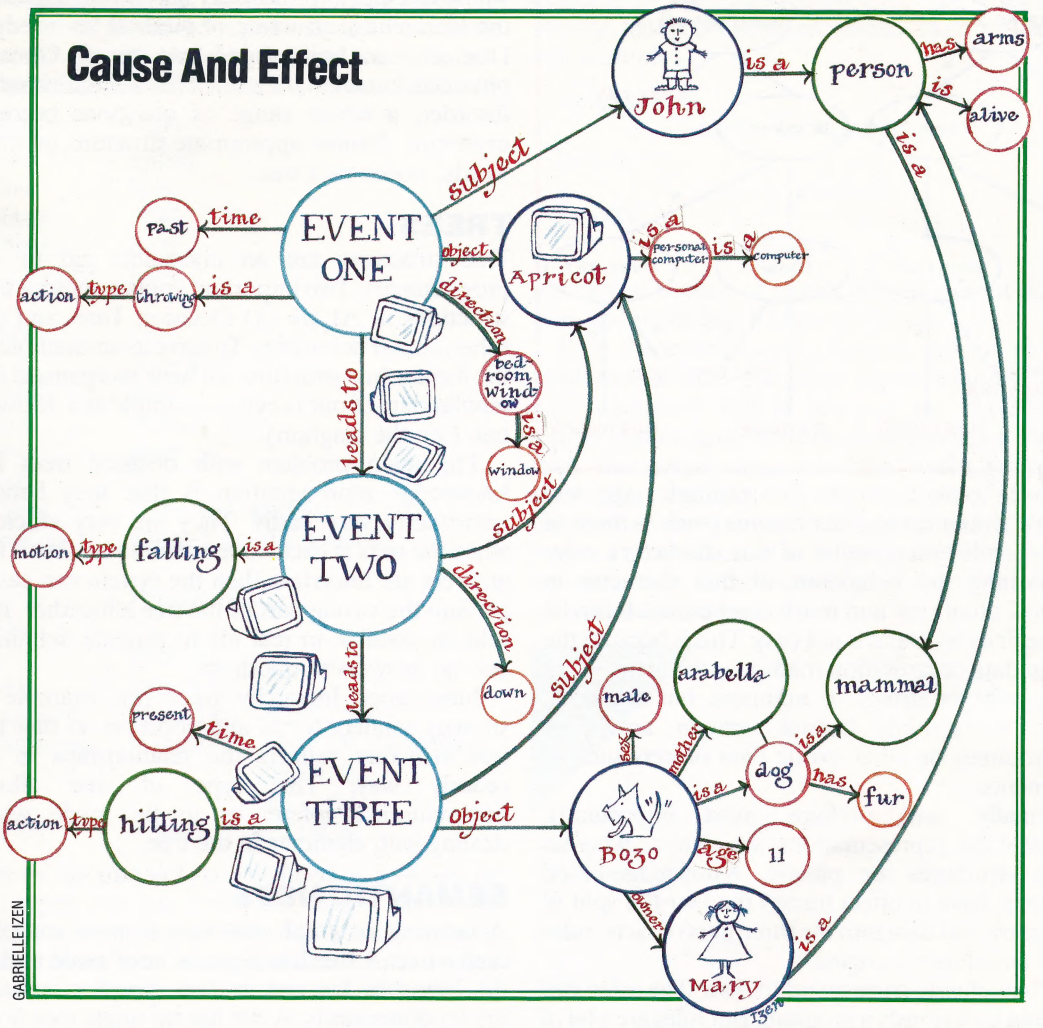
THE UNTHINKABLE Are games that portray nuclear war all that bad? INSIDE BACK COVER

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designers** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editor** Jon Kaye; **Contributors** Richard Forsyth, Marcus Jeffery, Bobby Pickering, Steve Colwill, Steve Cooke, Joe Pritchard, Steve Malone, Steven Vickers, David Mudd; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY CRISPIN THOMAS

IN THE KNOW



Semantic Sequence
Semantic nets are used to connect actions and objects in order to describe their relationships. The net shown here represents: 'John threw the PC out of the window. It hit Mary's dog.' The circles are the object nodes and the arrows are relationship arcs. From the semantic links given it could be deduced that, for example, 'A computer hit a furry mammal' or 'Something fell on Arabella's child'. In a real system the network would be far more extensive than the simplified one given here

In AI, information representation – modelling of the real world – is crucial. Getting it right means a working system. Getting it wrong, by mismatching the internal representation to the external world, frequently leads to total failure. We look at the different structures available to the AI researcher to represent items of data and their relationships.

AI programmers seek to mimic complex systems like human memory and reasoning, which is no easy task. This is one reason why they speak of 'knowledge representation' rather than 'data representation'.

Most programmers have a good idea of what is meant by data. But they are often sceptical about using the word 'knowledge' to describe information held in a computer. In the end, the difference between data and knowledge comes down to a difference in emphasis, arising from the nature of the problems that AI programmers encounter and the solutions they devise. It is useful to look at the four main components that make *knowledge* distinguishable from *data.*

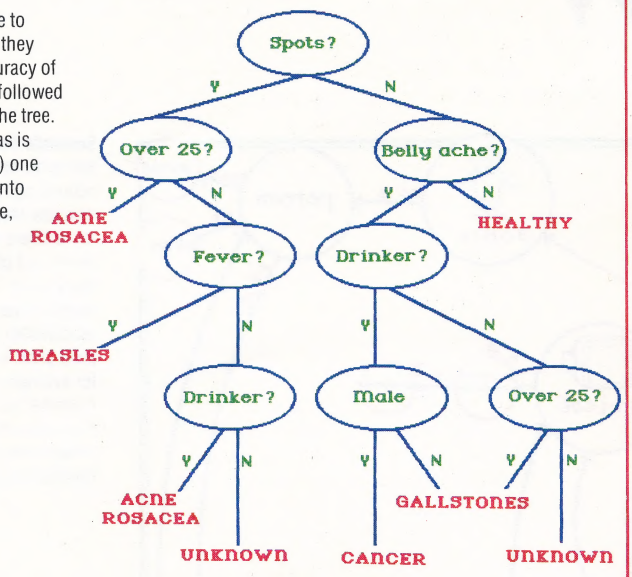
In the first place, knowledge representations need to be flexible rather than static. Knowledge must be encoded by structures that can shrink or grow as the program runs (using dynamic memory allocation) rather than by structures whose size and shape are fixed for the duration of the run. Unfortunately most dialects of BASIC do not provide dynamic data structures.

Secondly, knowledge representation requires multi-level or layered structures rather than single-level ones. For instance, lists may need to contain sublists, trees may contain subtrees, rules may refer to subrules, and so on.

Thirdly, knowledge representations are typically heterogeneous: they contain a mixture of data types. For example, you might want to set up a record describing a character in an adventure game. It would need to contain some strings (a

Pathways To Prescriptions

Decision trees are simple to build and use. However, they rely on the absolute accuracy of the information used to followed a certain route through the tree. If the data is uncertain (as is often the case in real life) one wrong answer can lead into the wrong area of the tree, resulting in a diagnosis that is wholly inaccurate



CAROLINE CLAYTON

name), some numbers (for example, age and height), pointers to other records (such as those of the friends and enemies of this character), rules describing the behaviour of this character in typical situations, and much else besides. It can be done in BASIC, but it is not easy. This is because the main data organisation method – the array – can only contain strings or numbers, but not both. And it certainly cannot contain arrays or subroutines or other exotic data objects such as elements.

Finally, and perhaps most importantly, knowledge representations are active, whereas data structures are passive. Knowledge-based systems have in effect turned the two-tier split of program and data into the three tiers of facts, rules and an inference engine.

Facts clearly correspond to data. The inference engine is obviously a program. But rules are a bit of both. They constitute *active data*. Having outlined some of the key differences between data and knowledge, let us look at some ways in which knowledge can be represented and stored.

ARRAYS

The array offers a simple method of representing knowledge. The only trouble is that it may prove too simple for many AI applications. Within the expert system context, for example, let us imagine a simplified medical diagnostic package with only four diagnoses (hypotheses) and seven symptoms (pieces of evidence). It would be possible to encode the knowledge of the system as a two-dimensional array (as shown).

This array format would be adequate to support a simple inference strategy; the evidence being collected one item at a time on a scale -1 (no) through 0 (maybe) to +1 (yes). When all the evidence has been output, it can simply be multiplied by the numbers in the corresponding rows for every column. The column with the highest total is then the most strongly supported hypothesis.

Problems would arise in trying to extend such a representation. It is inflexible: with 200 pieces of evidence and 100 disease categories the array would have to contain 20,000 cells, most of which would be empty. It is also too 'flat', in that it ignores the hierarchical structure of medical knowledge. Diseases can be grouped into types. Once a physician knows that a patient has a certain kind of disorder, a whole range of questions become irrelevant. A more appropriate structure, in other words, would be a tree.

TREES

Tree structures are an invaluable aid to AI programmers. Two important applications for tree structures in AI are (1) Decision Trees and (2) Inheritance Hierarchies. To serve as an example of the former tree structure, we have reorganised the tabular data of our previous example as a decision tree (see the diagram).

The main problem with decision trees for knowledge representation is that they handle uncertainty very badly. They are very efficient when the tests at each node are clear-cut, but if the answers are uncertain then the system can easily get into the wrong part of the tree altogether. It is seldom possible in real life to provide definitive yes/no answers to questions.

Inheritance hierarchy trees (an example is shown) connect terms and concepts so that the tree structure reflects the relationships in an orderly way. This type of tree allows commonsense inferences to be made when dealing with elements in the tree.

SEMANTIC NETS

A network or graph structure is more complex than a tree. A tree has a special 'root' node and all the branches fan out in one direction, usually drawn downwards. A net has no single root node and the links can point in any direction.

Semantic networks are a particular kind of graph structure that has found favour in AI for representing knowledge about language and about actions and events. In a semantic network the nodes represent objects and the arcs or links between them represent relationships.

The semantic net is a generalisation of a LISP construct that we met in the last instalment — namely, the property list (see page 1444). Both property lists and semantic nets encode information as attribute-value pairings linked to an object.

It is chiefly a matter of convenience; and, in fact, both property lists and semantic nets can be subsumed under the concept of *tuples*. An 'n-tuple', for example, is simply a grouping of 'n' objects. Object-attribute-value representations (like semantic nets and property lists) employ triples — that is, 3-tuples, such as:

Object	Attribute	Value
BOZO	IS-A	DOG

It is only a short step from the idea of representing knowledge as a collection of triples, to the notion of frames. The frame is a structure, much used in AI work, which contains a number of 'slots'. Each slot is like a field in a conventional record in a file, but the number of slots is not fixed.

Thus, frames are, in essence, 'n-tuples' where 'n' can vary. A framelike representation describing some of the information in the semantic net discussed earlier, with a few more facts added, is:

```
Name:          Event-3
Type:          Hitting
Time:          Now
Subject:       PC49
Object:        Bozo
Antecedent:    Event-2

Name:          Bozo
Is-a:          —> Prototype-Dog
Owner:         Mary
Type:          Animate-Agent
Sex:           Male
Age:           11
Parents:       (Ferdinand, Arabella)
Involved-in:   (Event-3, Event-7, Event-20 . . .

Name:          Prototype-Dog
Has:           Fur
Can:           (Bark, Bite, Hunt . . .)
Legs:          Default=4
Examples:      (Bozo, Fido, Arabella, Spot)
Dangerous:     IF small AND asleep OR Tail-wagging
                        THEN No
                        ELSE Yes
To-Do:         If Sleeping THEN Let-lie
```

Here we see that slots can be filled by a variety of data types: Age has a number, Parents has a list, Dangerous has a decision rule, Is-a has a pointer to another frame, and so on. The slots correspond to attributes in a property list or to arcs in a semantic net.

THE LOGIC OF REPRESENTATION

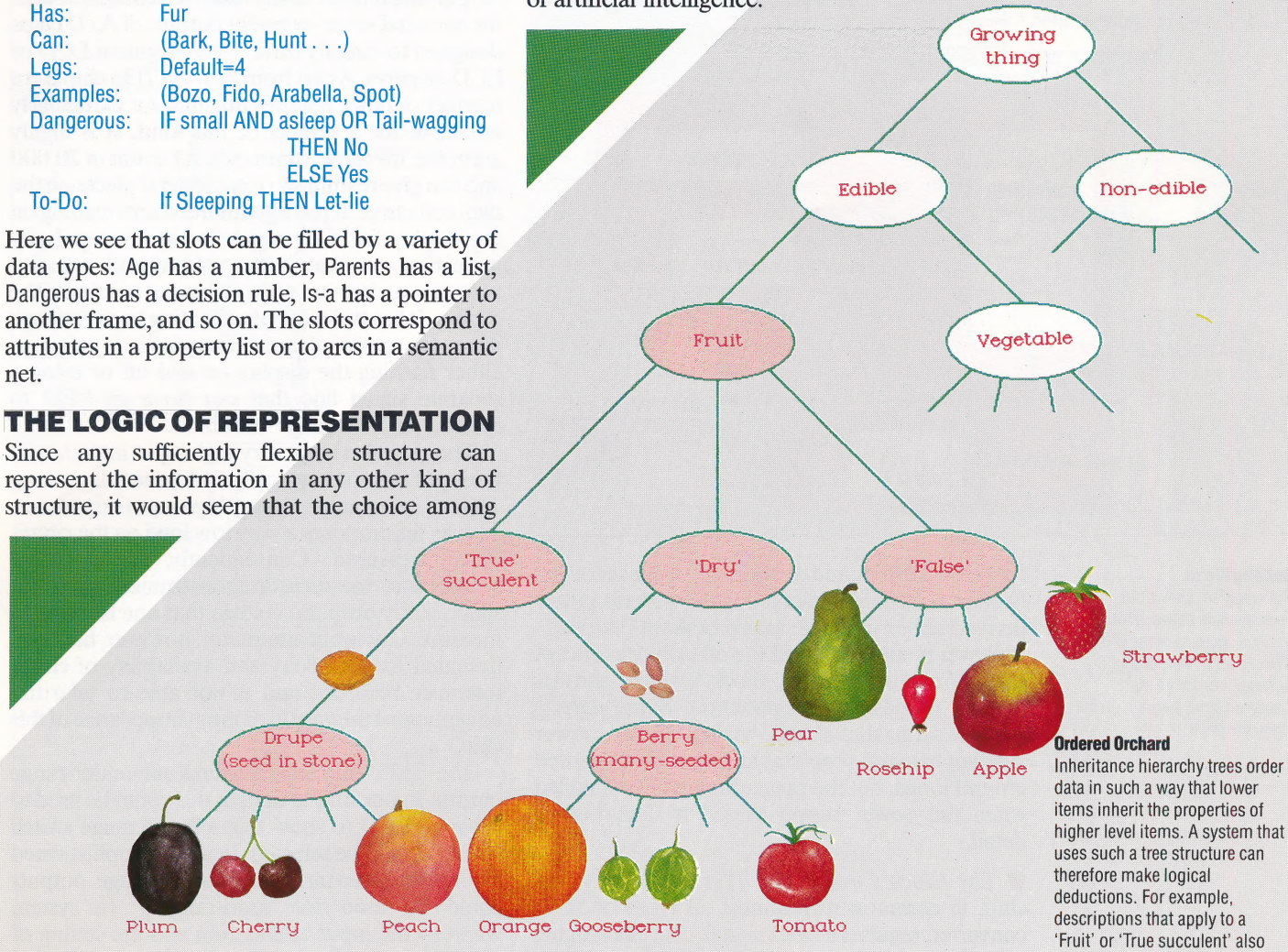
Since any sufficiently flexible structure can represent the information in any other kind of structure, it would seem that the choice among representations is primarily a matter of efficiency. However, some AI researchers dislike the relativism that this implies and seek to define an all-embracing formalism that is somehow more fundamental than the others. The formalism they choose is predicate logic. Logic as a representation language is the basis for PROLOG, which uses a scheme based on what are called 'horn clauses'. For example:

```
dangerous (X):-
        dog(X), not (safe(X))
safe (X):-
        small (X), asleep (X)
safe (X):-
        tail-wagging (X)
```

is a PROLOG version of one of the rules about dogs from the preceding frames structures.

In a theoretical sense, logic is indeed more fundamental than the other representations we have discussed. But in practice, the choice of representation is dictated by considerations other than theoretical elegance. The well-equipped knowledge engineer should be aware of the varieties of representation schemes, which have been successfully used in AI projects. Suiting the representation to the knowledge is one of the arts of artificial intelligence.

Sorting Out The Fruits



Ordered Orchard
Inheritance hierarchy trees order data in such a way that lower items inherit the properties of higher level items. A system that uses such a tree structure can therefore make logical deductions. For example, descriptions that apply to a 'Fruit' or 'True succulent' also apply to tomatoes and plums

ILLUSTRATIONS BY DAVID DAVIES

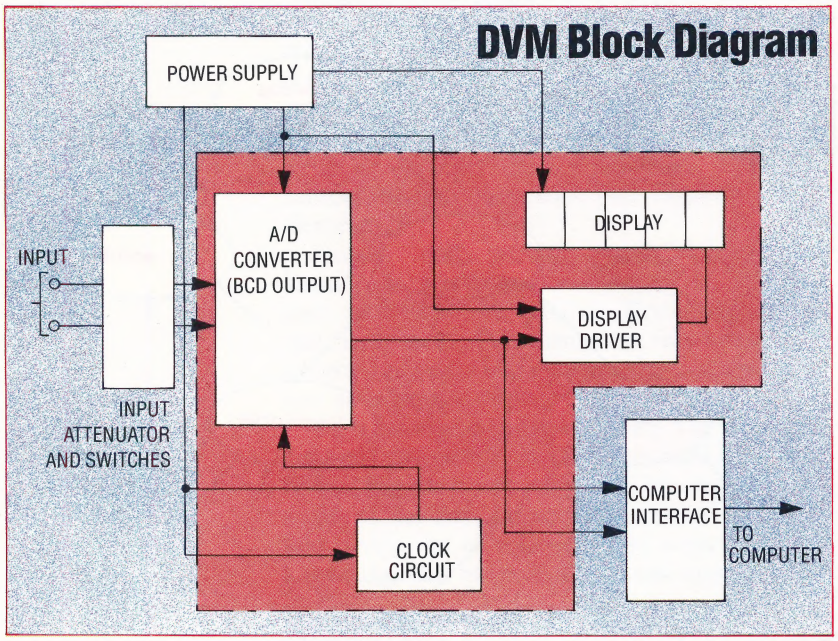
A BLOCK OFF THE OLD CHIP

In the previous instalment, we discussed some of the principles behind digital-to-analogue (D/A) and analogue-to-digital (A/D) conversion. Here we give an overview of the complete circuit of the digital volt meter we are designing.

The DVM circuit consists of seven sub-sections or blocks (as shown in the diagram). Let's consider each of these components in turn:

- *The Input Attentuator and Switches:* The input signal to be measured is fed by probe leads to the input terminals of the DVM via the input attenuator and switches block. The complexity of the attenuator and switches depends on how wide the range of values to be measured is (from microvolts to kilovolts, for example) and how many electrical units are to be measured (DC volts, AC volts, DC current, ohms, and so on). The circuit compromises involved, and the circuits themselves, will be described in detail later.

- *The Power Supply:* This block is self-explanatory. All that is required is +5v and −5v at fairly low current and this is easily achieved using two three-terminal voltage regulators with power derived either from the mains or from batteries.



Building Blocks
This diagram shows the relationships between the various sections of the DVM. The red area of the diagram corresponds to the more detailed circuit diagram on the opposite page

Power is supplied to all the other blocks except the attenuator/switches block (which consists entirely of passive components). Some care is needed with the actual wiring of the power supplies to avoid problems arising from unwanted ground loops, but the precautions will be detailed when the power supply circuit is described in detail.

- *The Clock Circuit:* The 7135 A/D converter chip, in common with almost all types of A/D converter, requires a clock signal. The clock signal is vital, but very simple (unlike the clock signals needed by some microprocessors) and can easily be generated by a couple of TTL (transistor-transistor logic) gates with feedback. We have opted for a circuit based on the versatile 555 timer chip. Although more sophisticated internally, the only external components needed are two resistors and a capacitor.

- *The Computer Interface:* This block is an optional extra. The basic DVM was designed as a stand-alone unit as a low-cost alternative to the expensive digital DVMs on the market. However, if you would like your computer to be able to take direct readings from the environment (in volts, ohms, degrees of temperature, or whatever), then this block can be added. It doesn't affect any of the other circuitry, and is strictly optional.

- *The A/D Converter:* This is really the heart of the DVM. It uses a single chip configured to have a basic sensitivity of two volts (or, to be precise, 1.9999 volts). Higher measured voltages are attenuated by the input attenuator so that the 7135 chip itself should never see an input voltage of more than two volts.

This chip has a number of advantages over other one-chip A/D converters. It converts the input voltage into a BCD (binary coded decimal) output that is more easily read by a computer than the encoded seven-segment outputs of A/D chips designed to directly drive seven-segment LCD or LED displays. Apart from this, the 7135 chip has a number of other features that make it particularly attractive for a project of this kind. It is highly accurate; the basic accuracy is ±1 count in 20,000 and can give readings to four decimal places on the two-volt range. It has a guaranteed zero reading on the display for a 0v input. It has 'auto-polarity indication', so that both positive and negative voltages can be read without having to reverse the probes. It has two ways of indicating an over-range input (an input of more than 1.9999 volts) — either flashing the display on and off or using a separate signal line that can drive an LED to indicate an over-range condition.

The chip also has a very high input impedance; the input current is typically 1pA (one trillionth, or 1×10^{-12}, of an amp). A very high input impedance such as this imposes a very low load on the circuit being measured. Consequently, the drain of current into the measuring instrument does not significantly drop the voltage that one is trying to measure. Our input attenuator, however, has been designed for simplicity and availability of close-tolerance resistors, and is not able to take full advantage of the very high input impedance of this chip.

The 7135 chip also features an under-range output. In our simple design, this could be used to drive an LED to show that a lower range switch setting should be selected. In a more sophisticated DVM, the over-range and under-range outputs could be used for 'auto-ranging' (a system whereby the input attenuation and the setting of the decimal point on the display are adjusted

automatically to accommodate the input signal).

The output voltage levels are direct TTL-compatible, and this makes interfacing to microcomputers comparatively simple. More importantly, a number of signals are provided specifically to simplify computer interfacing. These include a STROBE line to synchronise data transfer to external latches (registers that can hold data without the need for a constant input to be supplied), a RUN/HOLD input line that allows the computer to 'freeze' the reading or let the DVM free-run, and a BUSY line that can tell the computer whether the other outputs are valid or not. The four BCD output lines only have meaningful values at certain times during the A/D conversion process, and an active BUSY line tells you when the readings from the output lines are unreliable.

Above all, the 7135 chip is extremely versatile. Many of the available input and output signals can be ignored, but they are present and ready for use if a more sophisticated system is required.

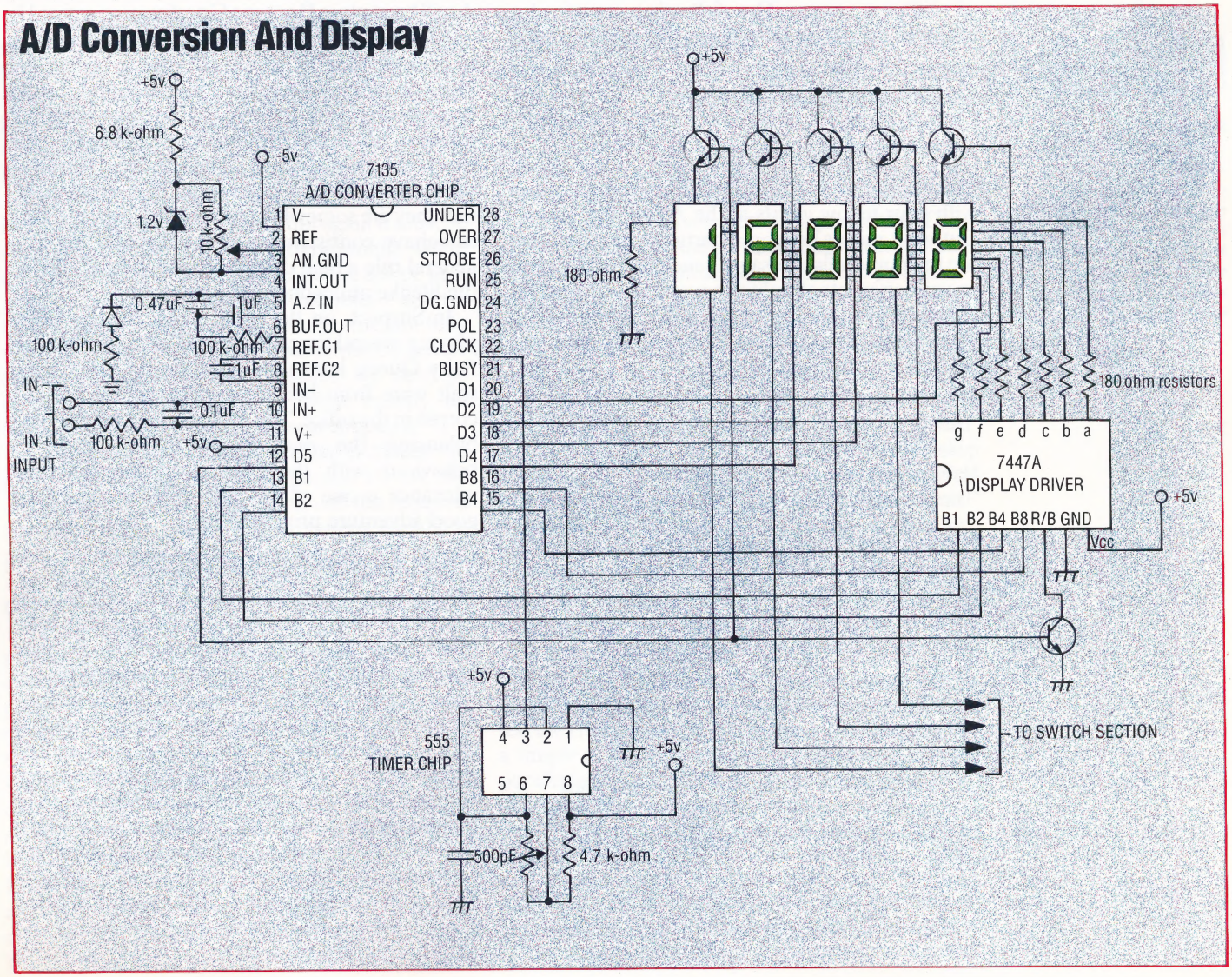
● *The Display Driver:* As the 7135 A/D converter provides a BCD output, it cannot be used directly to drive the display. Fortunately, a single, low cost chip is all that is needed to interface the A/D converter to a suitable display. The 7447A chip that we have opted for can convert a BCD signal into the right combination of seven signals needed to correctly illuminate a single seven-segment LED. The output of the 7135 is multiplexed. That is to say, the four BCD output signals are shared by each display digit but are valid only for one of the five digits at a time. At the same time, valid digit signals from the 7135 chip indicate which digit is being output in BCD at any moment. These valid digit signals are used directly to switch transistors, enabling one of the LEDs at a time. As the digit displays are switched in and out so rapidly, all five displays appear to glow simultaneously.

● *The Display:* LCDs are currently fashionable, particularly in battery-operated hand-held instruments. LEDs still, however, have a number of advantages, and we have opted for an LED display. LEDs are somewhat simpler than LCDs to interface, particularly where switching of the decimal points is concerned. LCDs require an AC back-plane signal, and extra logic circuitry is required in the DP (decimal point) circuit.

Converter Kernel

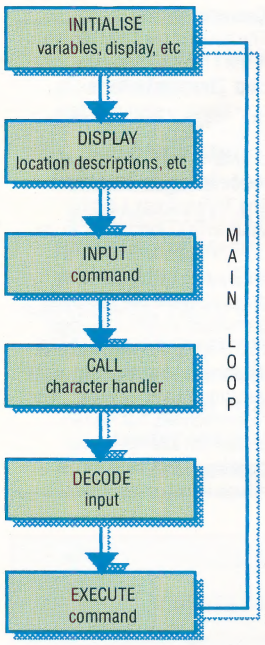
The basis of our DVM design is the 7135 A/D converter chip. This chip performs its operations in response to external clock signals provided by the 555 chip at the bottom of the diagram. The 7135 outputs a digital value in BCD form that can be used in conjunction with a multiplexing 7447A display driver chip and five switching transistors to light a 4½-digit LED display. Full construction details will be given in future instalments of the project. It is recommended that readers should not attempt to start building at this stage as the design is not yet complete

A/D Conversion And Display



KEVIN JONES

CONTROLLING THE CAST



CAROLINE CLAYTON

Strategic Position
This flow diagram of a typical adventure game shows one possible position for our character-handler routine. When incorporated into the program at this point in the flow of control, the routine will be executed every time the player presses ENTER after typing a command

The programming of player/character interaction in an adventure game can be quite complicated. We examine the possibilities, discuss the pros and cons and introduce two systems of character control – 'synchronous' and 'asynchronous'.

In our first instalment (see page 1441) we saw how the computer-controlled characters in Infocom's Suspect can move from location to location, apparently 'at will'. Movement, however, is one of the least important features of a computer-controlled entity and the program provides excellent examples of the more relevant player/character interaction in its command syntax. While playing Suspect, you can use commands that enable you to address characters in a number of different ways. The following inputs, for example, would all be accepted by the program:

Michael, where is Veronica?
Accuse Colonel Marston of murder
Call my lawyer
Johnson, tell me about youself

There is obviously scope, then, for communicating with your fellow guests at the Ashcroft mansion. But what makes the game particularly impressive are the numerous messages generated in response to your attempts at polite conversation. Although characters do tend to be slightly repetitive, they still manage to exhibit an unusual coherence and relevance in their replies:

Michael, tell me about the horse
"Lurking Grue is Veronica's prize jumper. He's really quite a beautiful animal."
Marston, tell me about the horse
"I don't know anything about it that would interest you."

Furthermore, not only can characters communicate with the player, they can also talk among themselves. Indeed, you are not likely to get very far in the game unless you linger in the ballroom and are party to conversations such as:

Michael joins in the discussion. "I recall a black stallion that went for a high price last year. It was probably over a hundred thousand." Colonel Marston glares at him. Colonel Marston says, "I have a good memory for figures. The top price last year was ninety-two thousand." Cochrane glances at Michael, feeling betrayed. "Nonsense", he says angrily . . .

To sum up, then, the 'people' in Suspect fulfil a number of the requirements we stipulated for truly interactive characters in the first instalment:

1. They can move from location to location.
2. They are addressable by the player and can respond meaningfully.

They can also pick up and drop objects — indeed one character, at one point in the game, actually goes so far as to hide an object inside another object! Finally, the characters can address you without being prompted. There are numerous instances of this during the game, the most obvious being when you are arrested by the detective (if you have failed to find the culprit).

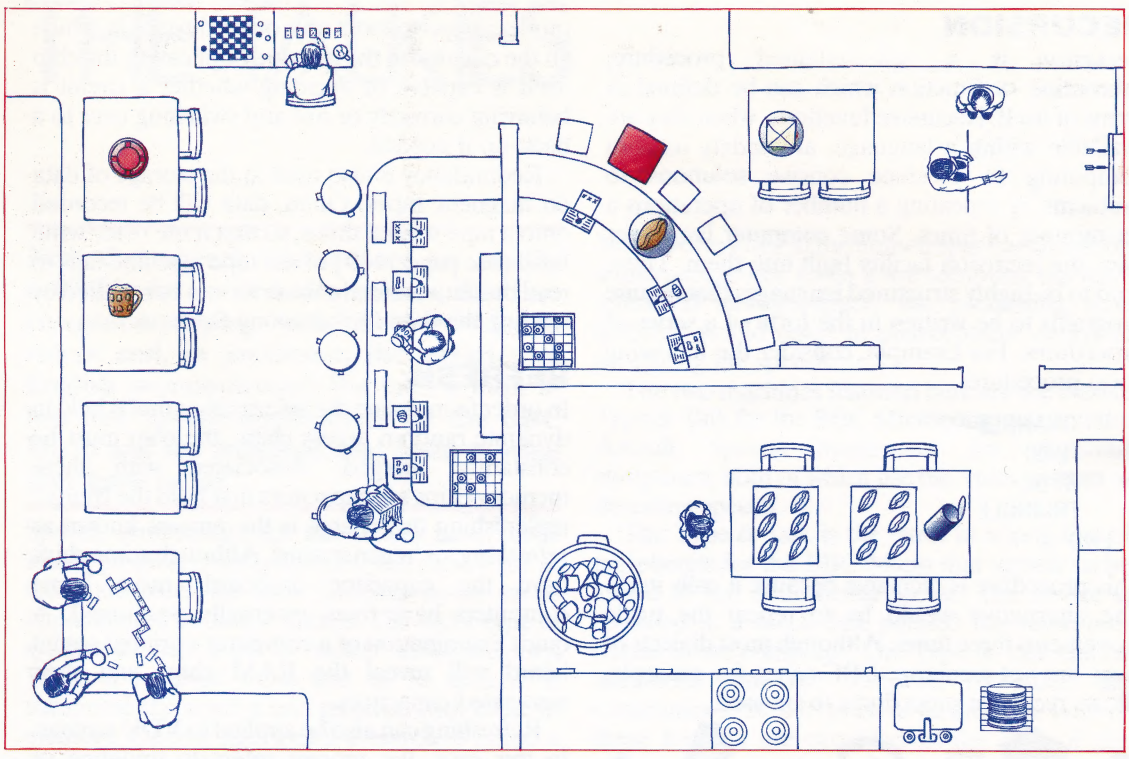
Before we go on to the next step – programming our own character-handling routine – it's worth briefly mentioning one drawback of designing a game that relies so much on the behaviour and actions of computer-controlled characters. The plot in Suspect, for example, requires that at a certain moment 'someone' will be killed, and that at certain other times during the game, characters will perform significant actions that, if observed or deduced by the player, enable the mystery to be solved. For this reason, the characters cannot behave completely randomly. This means that after running the program several times, the player soon comes to know that, for example, character A will be at place B at time C performing action D.

We'll see later that this requirement for people to perform specific actions in order to further the plot needs careful consideration when constructing an efficient character-handling routine. In The Hobbit, for example, the characters demonstrate a considerable degree of randomness in their behaviour. This might mean that they are somewhat constrained in their ability to behave consistently and intelligently, but as a general rule a high degree of randomness adds to the lifelike atmosphere of a program.

In Suspect, on the other hand, there is (after playing several times) a sense of *déjà vu* when the Fairy Queen leaves the ballroom to clean some spilt wine from her dress, only to end up as a corpse in the office. The programmer will find that balancing the randomness of a character's behaviour with the need for coherence and relevance is one of the most difficult aspects of good adventure programming.

TWO CONSIDERATIONS

Having taken a close look at Suspect, let's consider our own routines. The first decision that has to be made is a general one — how complex do we want our program to be? In designing an adventure, it is generally quite simple to decide on, say, how many locations you want to have, but when it comes to dealing with characters, there are almost endless combinations. A game like Suspect requires large amounts of data storage for the messages used by the characters alone — not to mention the routines to control them. Infocom deals with this problem by writing for disk-based systems only, loading in data when necessary. But for the majority of home users in Britain, disks are an expensive luxury and all programs must therefore run wholly in RAM.



KEVIN JONES

Local Operators
Our character-handler routine will be set in the infamous Dog and Bucket, an English pub notable for its draught beer and excellent home-made pasties. The layout of the locations used in the routine is shown here, together with the initial positions of some of the more important objects. In the next instalment, we will show how these are to be programmed into the routine, and add characters to our illustration as and when they are introduced

This isn't as limiting as it might appear, as we will see in the next instalment when we examine Melbourne House's Sherlock — a RAM-based game with some rather sophisticated character-handling. However, since it pays to keep things reasonably simple, we will design our program to handle a maximum of seven characters.

The second question that needs to be answered is not quite so straightforward, but again it must be dealt with before you start programming. There are two methods of controlling characters in a game — *synchronous* and *asynchronous* — and we have to choose which one to use. To explain fully what these terms mean, and what implications our choice will have for our program, we need to refresh our memories as to what exactly an adventure program does and how it does it.

You can see from the diagram that each time the player types in an input and presses ENTER, the computer will decode the input, call the required routine, update the necessary system variables, print messages, and then return for another input. The question we need to answer is this: where in this structure do we put our character-handler?

To get back to the terms just mentioned, a synchronous character system would execute the character-handler routine at a fixed stage in the program (as shown in the diagram). Here, the character-handler subroutine will be called each time the player presses ENTER. This is a synchronous system, and Suspect is a good example of this type of program. It has certain advantages for the player — for example, if you leave the keyboard to go and have a cup of tea, you know that you will not be attacked (perhaps) by another computer-controlled character while you are in the kitchen. By the same token, it does mean that the pace of the game tends to be too reliant on the player's inputs.

An asynchronous system, on the other hand, will usually be interrupt-driven and will pass control to the character-handler every so often, regardless of whether you are at the keyboard or not. This is the system used by The Hobbit and Valhalla where, if you sit back and watch the screen, you will see characters coming and going, leading their own lives quite independently. This system has the advantage of adding 'atmosphere' to a game, but it is also more difficult to program. In particular, you must be careful to ensure that there is no variable clash between your character-handler and the main program, otherwise you could end up with serious problems.

Using interrupt-driven routines from BASIC is very difficult, unless you happen to have an Amstrad or MSX micro; for this reason we will adopt the synchronous system for our routine. We will, however, design the program in such a way as to enable you to sit back and watch it running without the necessity of entering commands.

We are now in a position to start programming. Although the intention is to produce a transportable module that can be adapted to run with adventure games, such as our Digitaya and Haunted Forest programs (starting on page 766), we still need to provide an environment in which to test the character-handler. We will therefore set up a simple adventure with three locations in which our characters can move about, and with a number of objects for them to manipulate.

Since the program is designed to show off our interactive characters, we'll set the action in that well-known centre of conviviality — the English pub, or to be precise, the Dog and Bucket. What goes on there, and how you'll make it happen, will be the subject of the next instalment.

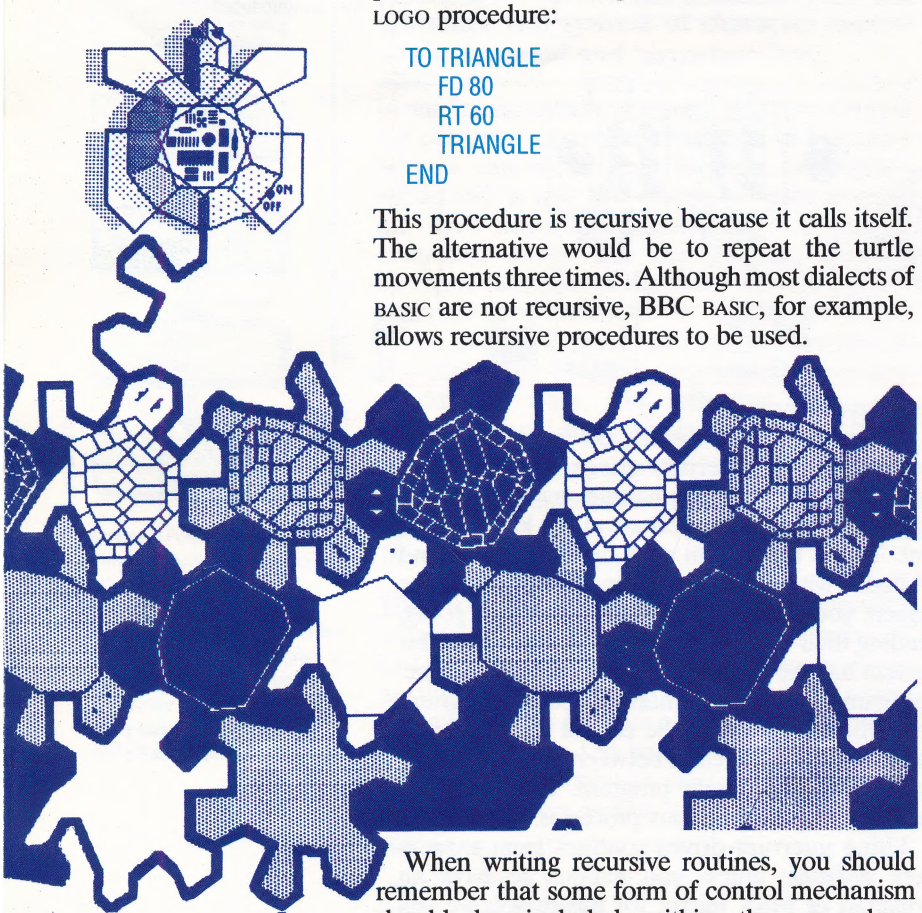
R

RECURSION

Recursion is a self-contained procedure, subroutine or function which can be defined in terms of itself. Recursive functions, when they are available within a language, are widely used in computing to produce concise solutions to problems by repeating a number of operations a set number of times. Some computer languages have the recursion facility built into them. These tend to be highly structured languages that require programs to be written in the form of a series of procedures. For example, consider the following LOGO procedure:

```
TO TRIANGLE
   FD 80
   RT 60
   TRIANGLE
END
```

This procedure is recursive because it calls itself. The alternative would be to repeat the turtle movements three times. Although most dialects of BASIC are not recursive, BBC BASIC, for example, allows recursive procedures to be used.



Recursion: See Recursion
This shows one of the primary uses of recursion in LOGO. Sequences of lines within a LOGO procedure can be recalled from within the procedure itself. This keeps coding to a minimum and means that variables within the procedure remain 'local'

When writing recursive routines, you should remember that some form of control mechanism should be included within the procedure, otherwise it will generate an infinite loop that can only be broken by halting the program.

REDUNDANCY

When designing or constructing a system, in order to allow for hardware error, often two or even three components are fitted that perform the same function. If one of them fails, its functions can be carried out by the back-up component. This feature is known as *redundancy.*

Although in the past redundancy has been mostly used with regard to back-up storage and power supplies, the concept is becoming increasingly important in the design of microcircuits. As these circuits are becoming even more miniaturised, the limits of the technology are being reached. The result is that a large proportion of modern microchips are rejected within the factory due to tiny manufacturing errors. However, many manufacturers are now producing processors and other devices in which all the circuits on the chip are duplicated; the chip itself is capable of deciding whether a circuit is behaving correctly or not and switching over to a back-up if needed.

Redundancy is also used in the storage of data on magnetic tapes. Often, data will be recorded onto a tape several times, so that if the read/write head (see page 1449) of the tape machine fails to read the data correctly, the error can be rectified by reading the data further along the tape.

REFRESH

In order to maintain the information that is held in dynamic random access chips, the chip must be constantly charged. Associated with these memory chips are capacitors that hold the charge; replenishing this charge is the process known as *refreshing*, or 'regenerating'. Although some chips have the capacitor on-board, most home computers have them externally positioned. A quick examination of a computer's printed circuit board will reveal the RAM chips and their associated capacitors.

Refreshing can also be applied to VDU screens. In this case, the process refers to updating or repeating the information that is displayed on the screen. This occurs when the electron beam in the cathode-ray tube has finished scanning, and the electrophosphorescent dots, which were excited by the electron beam, decay to a lower energy level as they emit light. Therefore, the electron beam needs continually to scan, or refresh, the screen in order to maintain a constant image (see page 1440).

REGISTER

A *register* is an address within memory (including the processor itself) which is used as a storage location for specialised data. Registers can be any length, although one, eight, 16 and 32-bit registers are the most common.

Because their contents usually require rapid access, registers are constructed to operate much faster than equivalent address locations in the memory. Registers are therefore made with fast-access 'bistable' transistors (see page 168).

Although registers are used by programmers to hold a variety of different data items, they are most often used for arithmetical and Boolean operations, such as in accumulators and counters.

RELATION

When examining a sequence of numbers, we can often say something about the properties that connect a pair of them. These properties are known as the *relation* between numbers. The kind of relations that we usually apply are: 'greater', 'less than' or 'equal to'. If there are only two operands to be considered, we can say that there is a 'binary relation' between them. Relations are found in many aspects of computing. Graphs, databases and array manipulation are just some of the applications of relation theory.

THE TALKIES

Now that speech synthesisers are readily available for most home micros, how are computers coping with their new-found voices and, in particular, the vagaries of English pronunciation? We take a look at two new speech synthesisers for the BBC Micro and the Amstrad, and listen to what they have to say.

Speech synthesisers for computers have made steady progress over the last few years and some of these advances are now filtering down to be implemented on home micros. The older type of synthesiser had a list of 'words' in its memory. When it received some text to be spoken, it compared the word it had received with those in memory. Once it had found a match, it called a routine that produced the series of sounds which, taken together, formed the required word. The modern speech synthesiser is somewhat more sophisticated. In these systems, the user can type in any phrase and the computer will 'speak' the words.

The problems involved in achieving successful and comprehensive speech synthesis using the text-to-speech approach are enormous. The sheer complexity of language is enough to defeat even the most sophisticated program. Take, for example, the word 'read'. This word can be pronounced as either 'red' or 'reed' depending on the context. Examples like this mean that microcomputers can, at best, take a word and make an attempt at the correct pronunciation. However, although they are by no means perfect, the newer systems can produce remarkable results.

The system works by reading a series of ASCII characters into a buffer. Then each word is taken and tested against a series of grammatical 'rules' which have been established by the programmer. Parts of the word will correspond to particular spellings for which a rule has been established. For example, one of these might be that if there is a vowel followed by a consonant followed by 'ie' or 'y' then the first vowel should produce a long sound. By implementing this rule, the word 'vary' will be pronounced correctly. However, the trouble with language, and the English language in particular, is that it is non-standard in its implementation. Thus, when confronted with a word like 'many', the speech synthesiser rule will determine that the correct pronunciation of the word is 'mayny'. Therefore, many programmers also include a list of common exceptions to the rules, although it is, of course, impossible to include every one in the limited space available on a microcomputer. Even if the computer did have unlimited storage space, the processing time involved would result in unacceptable delays in producing the speech.

The two machines featured here are the Namal Type & Talk for the BBC Microcomputer and the Amsoft Speech Synthesiser for Amstrad computers, both of which use the 'rules' system of decoding speech.

The Type & Talk is the latest in a long line of peripherals for the BBC Micro that appear to be aimed at the educational sector. The device is enclosed in a metal box painted the same buff colour as the computer. The Type & Talk can be interfaced with the computer by either the Centronics or RS423 ports as facilities for both of these formats are provided at the rear of the machine, together with a switch for toggling between the two. Also fitted on the back of the machine is a microjack socket that enables the synthesiser to be connected to an external speaker system.

The front of the Type & Talk system houses an on/off switch, a grille for the internal speaker and the volume control. Inside the device are two printed circuit boards, one containing the power transformer and speech synthesiser logic, while the attached daughter board contains the sound-amplifying circuits and speech encoder chip. Turning first to the main PCB, apart from the PIA chip and associated logic, the Type & Talk also has an eight Kbyte EPROM containing the dictionary firmware and grammatical rules. The processing required to decode information from the

NAMAL TYPE & TALK

PRICE
£171.35 inc VAT

SPEECH CHIP
Votrax SC01A

MODES
Direct text-to-speech conversion and phoneme modes

SOUND GENERATION
On-board amplifier and speaker

INTERFACES
RS423 and Centronics interfaces. Microjack socket provided for headphones or an external speaker system

STRENGTHS
Produces accurate pronunciation of a large number of words. Because the software is based in ROM, this leaves the computer free for other applications

WEAKNESSES
The synthesiser has a tendency to overheat, which causes corruption in the speech. It is also rather expensive

CRISPIN THOMAS

Speech, Speech
The Namal Type & Talk is a speech synthesiser designed for use with the BBC Micro. The device has its own processor and software held on ROM. Obviously designed for the educational market, in which the BBC Micro is the dominant computer, the machine is robustly constructed with excellent software, and provides an informative and entertaining introduction to speech synthesis

AMSOFT SPEECH SYNTHESISER

PRICE
£29.95

SPEECH CHIP
SP0256

MODES
Text-to-speech conversion and allophone modes

SOUND GENERATION
Twin speakers, which although they can be stereo produce mono reproduction. Volume control fitted

INTERFACES
Expansion bus which fits into the floppy disk port of the computer, with a flying lead to the hi-fi port. Twin microjacks for the speakers

STRENGTHS
A genuine text-to-speech conversion at a reasonable price

WEAKNESSES
The list of grammatical rules is not as comprehensive as the Type & Talk. Suffers from hardware restrictions imposed by the computer

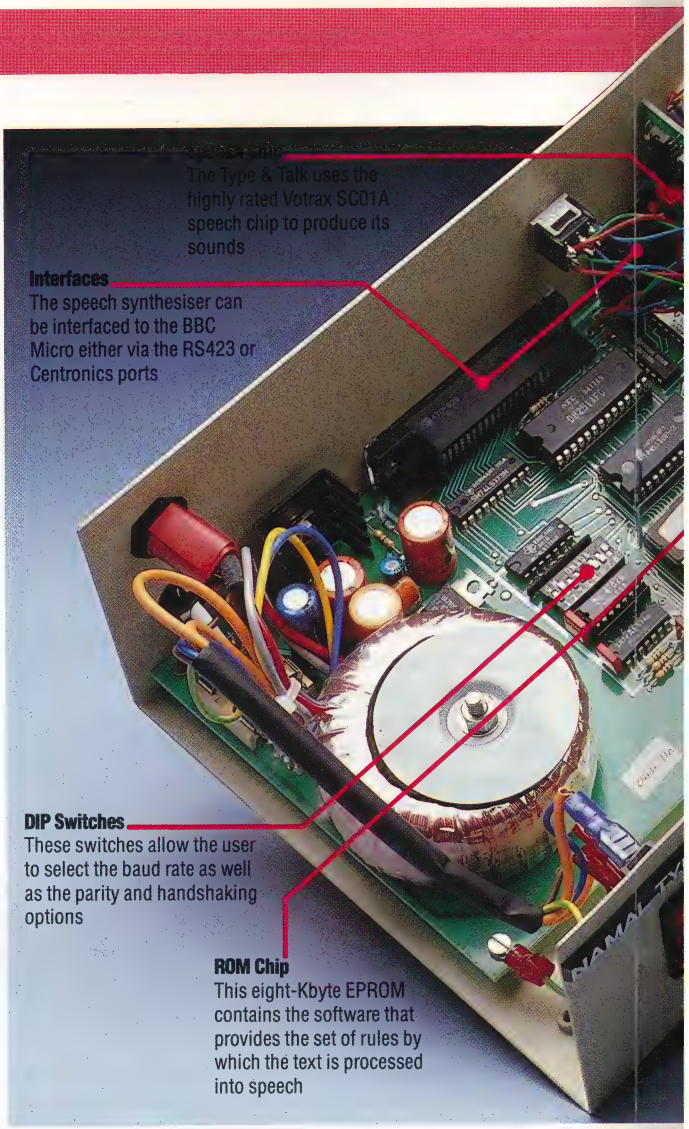
computer before passing it to the speech synthesiser chip is performed by an NEC D780C processor (which is essentially a Z80 chip). To store the incoming information before it is processed, the Type & Talk has two Kbytes for RAM on-board which act as a buffer. Also built on to the mother board are eight DIP switches which control the baud rate at which incoming information is handled, and other controls for 'handshaking' and parity selection. At least one third of the main board is taken up with the power transformer, which the manufacturers have chosen to include on-board.

While it is always more convenient to have the power supply within a machine to reduce the number of trailing wires, the problems that accompany such a move have not been entirely overcome in this case. When the Type & Talk has been left on for a few hours, the machinery shows signs of overheating, thus affecting the quality of speech produced. The daughter board contains the sound amplifier and a 'preset potentiometer' fitted with a screw-top that allows the user to adjust the rate at which speech is passed through the speaker. The actual speech synthesis is carried out by the highly rated Votrax SC01A speech chip, which is found on many other computer speech synthesisers.

Small Talk
The Amsoft Speech Synthesiser is much more modestly priced than the Type & Talk and yet uses the same 'rules' technique as the more expensive machine. Software is LOADed from cassette and the text is converted into speech patterns via the computer's own processor. Although not as ambitious as the Type & Talk, the Amsoft device is intended as an entry machine for the home user

Once the system has been connected, producing speech from the device is extremely simple. When the Type & Talk is switched on, the device says Ready Master, indicating that it is ready for use. As the computer receives signals from the synthesiser in the form of ASCII codes sent via the printer drivers, the computer must be configured to produce a printer 'echo' of whatever is typed to the screen. This is done on the BBC Micro by either typing VDU 2 or simply sending a CTRL B character to the Type And Talk. From then on, any phrases typed on to the screen will be spoken by the speech synthesiser.

We have already seen both how the Type & Talk system decodes words received from the computer as well as the limitations imposed by hardware

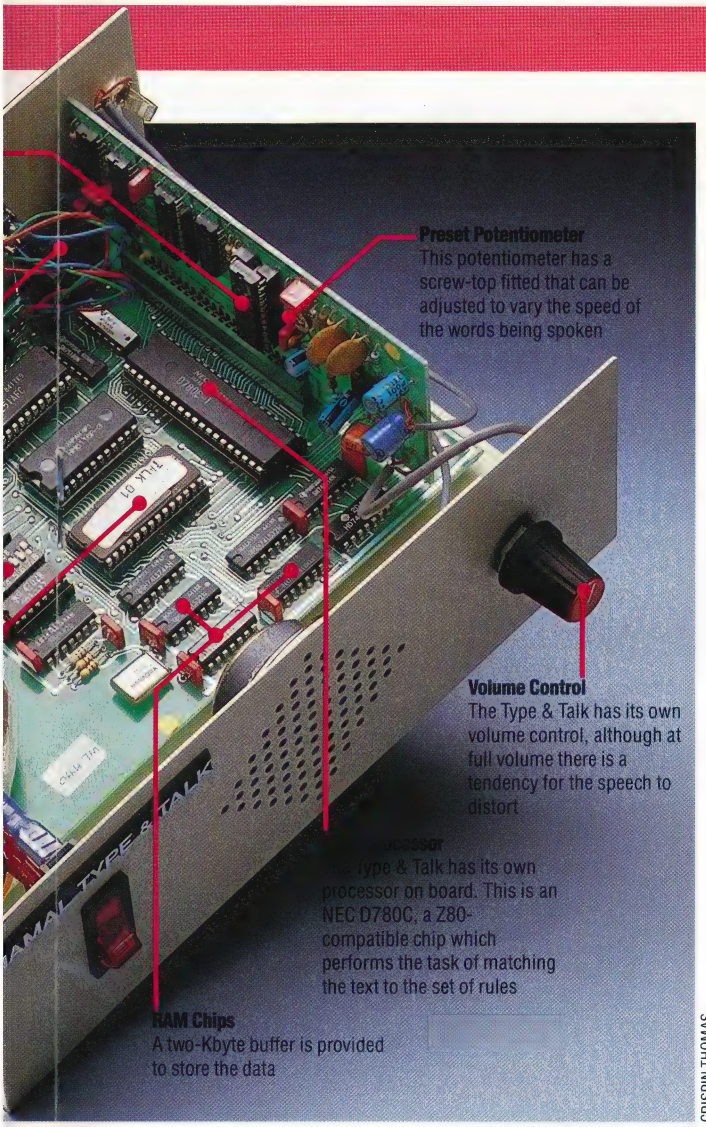

The Type & Talk uses the highly rated Votrax SC01A speech chip to produce its sounds
Interfaces
The speech synthesiser can be interfaced to the BBC Micro either via the RS423 or Centronics ports
DIP Switches
These switches allow the user to select the baud rate as well as the parity and handshaking options
ROM Chip
This eight-Kbyte EPROM contains the software that provides the set of rules by which the text is processed into speech

Speak Easy

As opposed to reading, when we can recognise different words by the way they are spelt, the recognition of spoken words is dependent on separate and distinct sounds, or 'phonemes.' While the grouping of several phonemes represent the 'bulk' of a word, slight differences and modifications (according to the context and place in which the words are spoken) are made with 'allophones'. For example, the phoneme CH is used in both 'chip' and 'batch', but the pronunciations are subtly different. The allophone thus alters the phoneme CH slightly to make recognition of the words easier.

As can be seen by the following example, the phonetic spellings of words often bear little relation to the ways in which we write them:

THV ER PA0	K W I K PA0	B R AW2 AW2 N PA0
THE	QUICK	BROWN
F O2 O2 K S PA0	D J UH1 M P T PA0	O V E R PA0
FOX	JUMPED	OVER
THV ER PA0	L A Z I PA0	D O G STOP
THE	LAZY	DOG


Preset Potentiometer
This potentiometer has a screw-top fitted that can be adjusted to vary the speed of the words being spoken
Volume Control
The Type & Talk has its own volume control, although at full volume there is a tendency for the speech to distort
RAM Chips
A two-Kbyte buffer is provided to store the data

CRISPIN THOMAS

restrictions. Thus to obtain the correct pronunciation of a word, it is often necessary to alter the spelling, and often a lot of ingenuity is required before the correct enunciation is achieved. For example, the word 'bough' is interpreted by the firmware in the same way as 'cough' and is thus pronounced 'boff'. Using the alternative spelling 'bow' does not help either, as this is interpreted to rhyme with 'low'. The correct pronunciation is achieved with the spelling 'bou'.

Emphasis on particular syllables can be obtained by sending control characters between the words to be pronounced. These control characters are distinguished from ordinary ASCII characters by preceding them with !. An example of this is the word HEL!InLO, where I stands for 'intonation' and n is a number between 0 and 3.

Another use of control characters that provides the user with far greater control over pronunciation is !P, which puts the speech synthesiser into phoneme mode. Phonemes are a series of characters that correspond to all the sounds made in English. (You'll see phonetic spellings in dictionaries – they are usually in brackets following a bold entry.) By sending a series of phonemes, any word with any accent can, in theory, be constructed.

The Amsoft Speech Synthesiser is a very different machine. This is a single unit that plugs into the floppy disk port of Amstrad computers. The text-to-speech conversion is performed by the Amstrad's own processor, with the dictionary and rules supplied by cassette-based software. Commands are sent to the speech synthesiser in a somewhat different way to that of the BBC Micro. There are nine commands that allow the synthesiser to be operated from BASIC, which, like the Amsoft disk operating system, are implemented as resident system extensions (RSXs). The problem with this system is that before parameters can be passed to the RSXs, they first have to be converted into strings. For example, |SAY(with |ECHO making up the two direct text-to-speech conversion commands) requires that before a sentence can be passed to the speech synthesiser, it first has to be implemented as a string. Luckily, the |ECHO command, which reproduces screen messages via the synthesiser, has parameter-passing built in.

QUALITY COMPARISON

Neither the sound nor the text-to-speech conversion quality of the Amsoft Speech Synthesiser is as good as that of the Type & Talk. This is not surprising, perhaps, as the Amsoft device is considerably cheaper than the BBC Micro peripheral. The Amsoft Speech Synthesiser has far fewer rules which are less rigorously implemented than on the Type & Talk. The result is that many words that are correctly pronounced on the Type & Talk are mis-spoken on the Amsoft synthesiser. For example, 'able' is pronounced 'abble' and 'cough' is pronounced 'cowf'. It is possible, however, to overcome these difficulties with imaginative spelling.

The other main command implemented on the Speech Synthesiser is |APHONE. This command, like the !P on the Type & Talk system, allows words to be built up from their basic sound components. However, the Amsoft system uses allophones as the 'building blocks' instead of phonemes (allophones being the constituent sounds of phonemes, although in practice there is little difference). The |APHONE command is followed by a series of numbers, each of which corresponds to a different sound. This system lacks the immediacy of Type & Talk's phoneme system and means that the user has constantly to be looking up the relevant number of the sound that is required, but it's fairly easy to get used to.

Many of the problems involved in speech synthesis have now been largely overcome, despite the fact that speech synthesisers still invariably sound like Daleks. But difficulties still remain in devising a program to take account of all the variations in speech that occur. Current technology is clearly not equal to the task, although the emerging CD-ROM (compact disk ROM) may go some way towards solving the 'rule' storage problems. However, both these packages represent the current state-of-the-art and give a good grounding in the technology around which future systems will be based.

ROLLING STONES

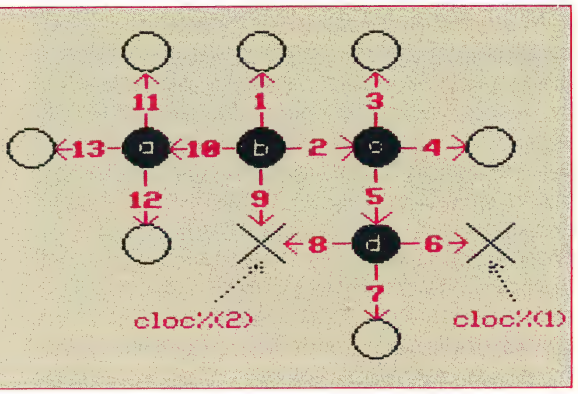
In choosing to make a move in Go, you are faced with a considerably greater number of possibilities than you are when deciding to make a move in a game of chess. Here we program the first in a series of modules for our Go game that allows the computer to 'decide' where to place its next stone.

In this instalment we'll begin the real artificial intelligence part of the program, where we 'teach' the computer how to choose reasonable moves. The first 'computer-move' routine will enable the computer to decide whether or not any particular board group is in trouble. When this has been determined, we can show the computer how to select a move which is likely either to save the group, if it is its own, or attack and capture an opponent's group.

Before we begin tackling this problem, however, let's look at the way in which most game programs choose moves (see also page 301). In the game of chess, for example, there are reckoned to be — on average — approximately 30 possible moves form any given board position. Taking this as a basis, if a computer were to try every possible move, then it would have to examine 30 possible positions, deciding how good each is. When trying to decide the merit of each move, you will, of course, have to check which moves an opponent can make in response to your own — a move is obviously no good if the opponent can immediately checkmate you. If we assume that the computer is going to examine all the moves which its opponent can play from the 30 positions that the computer can initially reach, then it is necessary to examine somewhere in the region of 900 (30×30) possible positions. Taking this a stage further, we could look at all the computer's possible replies, giving approximately 27,000 positions to be evaluated, followed by 810,000 positions at the next level, and so on. This 'look-

Condition Critical

The group evaluation routine in the program considers a group to be in danger if it has only one or two liberties. The diagram here shows the order in which the black group is evaluated, starting from stone b. Two array elements, cloc%(1) and cloc%(2), are used to store the positions of any liberties encountered as, if the group is in danger, the computer will probably want to play on one of these points



Module Four

BBC Micro:

```
   80 move%=move%+1:PROCblack_move
  240 DIM capture%(2),cloc%(2),tloc%(2)
 1360 atari1$="      ":atari2$="      ":T$
="***"
 2530 :
 2540 DEF PROCblack_move
 2550 atari1$="      "
 2560 location%=0
 2570 PROCgroup_evaluation:T$="GP "
 2630 IF location%=0 THEN ENDPROC
 2640 PROCmake_move(location%,black%)
 2650 PROCmessage(23,6,"")
 2660 ENDPROC
 2670 :
 2680 REM*****************************
 2690 :
 2700 DEF PROCgroup_evaluation
 2710 LOCAL C%,L%,P%,Q%,S%,hi,score
 2720 FOR P%=17 TO 255
 2730   C%=board%?P% AND colour%
 2740   IF C%=0 THEN 2850
 2750   PROCcount(P%,C%) : IF clib%>2 TH
EN 2850
 2760   tloc%(1)=cloc%(1)
 2770   tloc%(2)=cloc%(2)
 2780   L%=clib% : S%=cstn%
 2790   FOR Q%=1 TO L%
 2800     IF FNlegality(tloc%(Q%),black%
)<>0 THEN 2840
 2810     IF L%=2 AND clib%<3 THEN 2840
 2820     score=(8*S%/L%-clib%+2*L%)
 2830     IF score>hi THEN hi=score:loca
tion%=tloc%(Q%)
 2840     NEXT
 2850   NEXT
 2860 ENDPROC
 2870 :
 2880 REM*****************************
 4060 cloc%(1)=0 : cloc%(2)=0
 4280 IF clib%<3 THEN cloc%(clib%)=P%
```

Amstrad CPC 464/664:

```
80 mve%=mve%+1:GOSUB 2540
240 DIM capture%(2),cloc%(2),tloc%(2)
1360 atari1$="      ":atari2$="      ":T$=
"***"
2530 :
2540 REM **** black move routine ****
2550 atari1$="      "
2560 location%=0
2570 GOSUB 2700:T$="GP ":REM group evalu
ate
2630 IF location%=0 THEN RETURN
2640 mmp%=location%:mmc%=black%:GOSUB 36
30:REM make move
2650 mp%=25:mm%=6:o$="":GOSUB 2160:REM m
essage
2660 RETURN
2670 :
2680 REM ************************
2690 :
2700 REM group evaluation routine
2710 hi%=-9999
2720 FOR p%=17 TO 255
2730 c%=PEEK(board+p%) AND colour%
2740 IF c%=0 THEN 2850
2750 cp%=p%:cc%=c%:GOSUB 4040:IF clib%>2
 THEN 2850
2760 tloc%(1)=cloc%(1)
2770 tloc%(2)=cloc%(2)
2780 gl%=clib%:gs%=cstn%
2790 FOR q%=1 TO gl%
2800 lp%=tloc%(q%):lc%=black%:GOSUB 3890
:IF ll%<>0 THEN 2840
2810 IF gl%=2 AND clib%<3 THEN 2840
2820 score=(8*gs%/gl%-clib%+2*gl%)
2830 IF score>hi THEN hi=score:location%
=tloc%(q%)
2840 NEXT q%
2850 NEXT p%
2860 RETURN
2870 :
2880 REM **********************
4060 cloc%(1)=0:cloc%(2)=0
4280 IF clib%<3 THEN cloc%(clib%)=sp%
```

Commodore 64:

```
80 MOVE%=MOVE%+1:GOSUB 2540
240 DIM CAPTURE%(2),CLOC%(2),TLOC%(2)
1360 A1$="      ":A2$="      ":T$="***"
2530 :
2540 REM BLACK-MOVE ROUTINE
2550 A1$="      "
2560 LOCAT%=0
2570 GOSUB 2700:T$="GP "
2630 IF LOCAT%=0 THEN RETURN
2640 MP%=LOCAT%:MC%=BLACK%:GOSUB 3630
2650 MP%=24:MM%=6:O$="":GOSUB 2160
2660 RETURN
2670 :
2680 REM*********************************
2690 :
2700 REM GROUP-EVALUATION ROUTINE
2710 HI=-9999
2720 FOR P=17 TO 255
2730 C%=PEEK(BOARD+P) AND COLOUR%
2740 IF C%=0 GOTO 2850
2750 CP%=P:CC%=C%:GOSUB 4040:IF CLIB%>2
GOTO 2850
2760 TLOC%(1)=CLOC%(1)
2770 TLOC%(2)=CLOC%(2)
2780 BL=CLIB%:BS=CSTN%
2790 FOR Q=1 TO BL
2800 LP%=TLOC%(Q):LC%=BLACK%:GOSUB 3890:
IF LL%<>0 GOTO 2840
2810 IF BL=2 AND CLIB%<3 GOTO 2840
2820 SCR=(8*BS/BL-CLIB%+2*BL)
2830 IF SCR>HI THEN HI=SCR:LOCAT%=TLOC%(
Q)
2840 NEXT
2850 NEXT
2860 RETURN
2870 :
2880 REM*********************************
4060 CLOC%(1)=0:CLOC%(2)=0
4280 IF CLIB%<3 THEN CLOC%(CLIB%)=SP%
```

Sinclair Spectrum:

```
  80 LET move=move+1: GO SUB 2540
 240 DIM c(2): DIM i(2): DIM j(2)
1360 LET x$="      ": LET y$="
  ": LET t$="***"
2530:
2540 REM black-move routine
2550 LET x$="      "
2560 LET location=0
2570 GO SUB 2700: LET t$="GP "
2630 IF location=0 THEN  RETURN
2640 LET mmp=location: LET mmc=b
lack: GO SUB 3630
2650 LET mp=21: LET mm=7: LET o$
="": GO SUB 2160
2660 RETURN
2670:
2680 REM **********************
2690:
2700 REM group-evaluation routine
2710 LET hi=-9999
2720 FOR p=17 TO 255
2730 LET c=PEEK (board+p)
2735 IF c>3 THEN  LET c=c-4: GO
TO 2735
2740 IF c=0 THEN  GO TO 2850
2750 LET cp=p: LET cc=c: GO SUB
4040: IF clib>2 THEN  GO TO 2850
2760 LET j(1)=i(1)
2770 LET j(2)=i(2)
2780 LET gl=clib: LET gs=cstn
2790 FOR q=1 TO gl
2800 LET lp=j(q): LET lc=black:
GO SUB 3890: IF ll<>0 THEN  GO T
O 2840
2810 IF gl=2 AND clib<3 THEN  GO
 TO 2840
2820 LET score=(8*gs/gl-clib+2*gl)
2830 IF score>hi THEN  LET hi=sc
ore: LET location=j(q)
2840 NEXT q
2850 NEXT p
2860 RETURN
2870:
2880 REM **********************
4060 LET i(1)=0: LET i(2)=0
4280 IF clib<3 THEN  LET i(clib)
=sp
```

ahead' evaluation can be shown diagramatically by a 'game-tree' (similar to a family tree) where the branches from each node show the possible moves from that position. This concept has been fully discussed in our series on artificial intelligence (see pages 1268 and 1281). Horrendous though these numbers may appear they can be reduced somewhat using a variety of techniques — particularly the 'alpha-beta algorithm' — and they are manageable by high-speed computer.

The average number of moves available from a given position in the game of Go (on a 19 by 19 board) is estimated to be around 200. On our 15 by 15 board this number is likely to be around 125. If we start our standard look-ahead technique, the exponential growth gives 15,625 possibilities for the opponent's replies, 1,953,125 for the computer's second move, 244,140,625 positions from this, and so on. Fast computers or not, you can't really expect them to search through this number of possibilities in a reasonable period of time. Furthermore, it is not unusual for masters of the game to examine possibilities up to 20 or 30 moves ahead in order to decide the outcome of a particular battle.

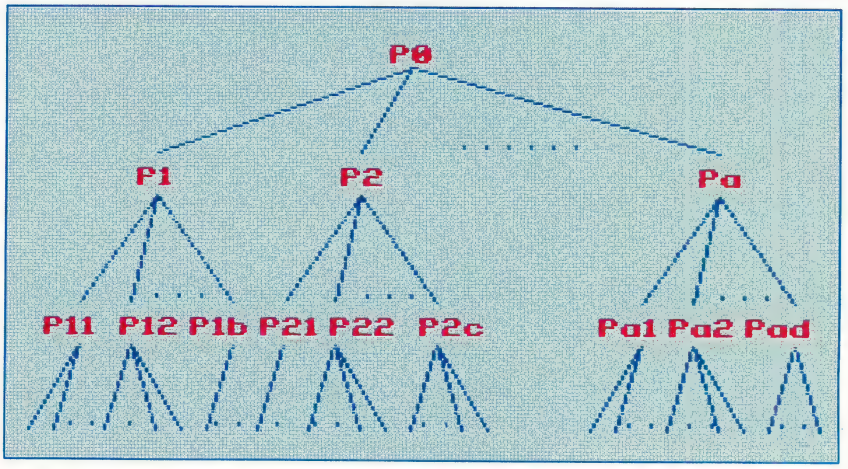
AN ALTERNATIVE STRATEGY

Clearly, it is futile to program our game of Go to determine a move by using 'brute force' look-ahead techniques. This is one of the major reasons why Go is so difficult to program. An alternative strategy is to develop a series of routines that will examine the present board position in the hope of, being able to choose some likely moves. There will eventually be six evaluation routines, and evaluation will terminate as soon as any one of the routines finds a feasible move. Consequently, the routines have to be called in the correct order. A variable, location% (initially zero), is set to the appropriate board position if a routine finds a move. Our main black_move calling routine will then be of the form:

```
LET location%=0
CALL                          the first evaluation
                                    routine
IF location%=0 THEN CALL the second evaluation
                                    routine
```

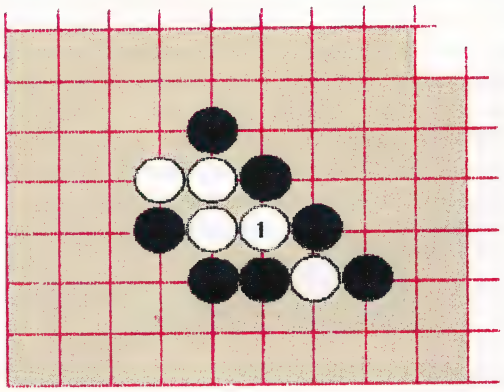
Go For Growth

Many games of skill, such as chess, use look-ahead game trees to decide which move to make. The nature of Go, however, means that around 200 new descendent branches have to be considered just to look one move ahead; looking two moves ahead entails checking 40,000 moves, and so on. If we compare this rate of growth with chess, we can see why producing a computer program that can play Go to a high standard is so difficult

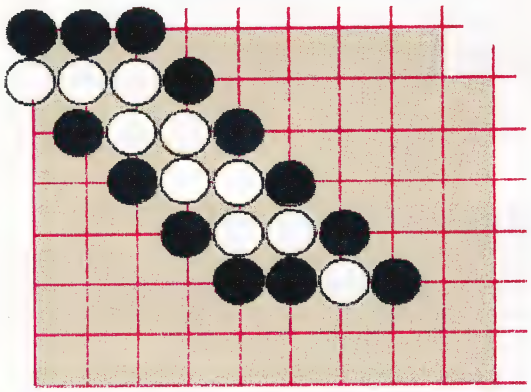


SHICHO

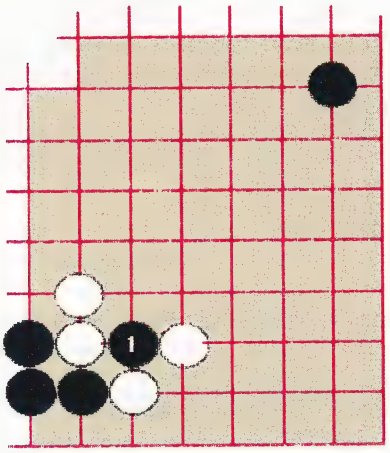
Dangerous Diagonals
White stone 1 is attacked by Black and, rather than sacrifice this stone, White tries to expand diagonally upwards and to the left in an attempt to avoid capture



Out Of Room
White has made a big mistake in trying to flee. Eventually the expanding white group will reach the edge of the board where it runs out of room and can be captured by Black. Rather then flee, White should have been prepared to surrender the original stone when first attacked. Diagonal zig-zag patterns formed during this type of play are known as SHICHO



Shicho Breaker
In this situation White might be tempted to try to capture the black stone marked 1, assuming that if Black tries to flee, a shicho will be formed and Black will run out of room at the right-hand edge of the board. However, as Black flees diagonally up and right it will eventually join with the single black stone, breaking the shicho and allowing Black to escape capture



CAROLINE CLAYTON

```
IF location%=0 THEN CALL the third evaluation
                         routine
      .        .        .
      .        .        .
      .        .        .
IF location%=0 THEN CALL the Nth evaluation
                         routine
IF location%=0 THEN cannot find any moves to play
```

As presented here, PROCblack_move only calls one evaluation routine, namely PROCgroup_evaluation, and if this doesn't find a move (if location% is still zero) then the routine simply terminates. Therefore, you can't as yet expect the program to play a proper game, although the computer will try to defend a group if you attack it.

The 'group evaluation' routine wil always have the highest priority in PROCblack_move, so it will always be checked first. Its purpose is to examine all the groups on the board and count their liberties. If any group has only one or two liberties, then it is considered to be in a critical situation (that is, only one move from 'Atari'), so the computer will either try to defend a computer group or attack an enemy group. The first action of the routine is to count the number of liberties in each group and save the positions of the liberties if the group only has one or two. We want to save these positions because our move is bound to be in one of these positions if we wish to extend the group to save it, or, if attacking, surround the group to capture it.

In a previous instalment (see page 1425), we pointed out that the PROCcount routine counts not only the stones in any group but also the number of liberties. By adding line 4060 to PROCcount and line 4280 to PROCsearch, the liberty count will be stored in the array cloc%(2).

GROUP EVALUATION

PROCgroup_evaluation can now be defined. The loop P% checks every position on the board. If a group is found, then its liberties are counted and if less than three, then the group is critical, so the loop Q% is entered. This checks the legality of placing a stone on the liberty(s), and if legal, the move is assigned a score. This scoring function (in line 2820) was hit upon mainly by trial and error, and is not necessarily the best — you might like to try experimenting. Notice that as a by-product of calling FNlegality, clib% and cstn% now contain the number of liberties and stones in the group assuming the move is played. Therefore, the original values of clib% and cstn% have been saved in L% and S%.

For simplicity, PROCgroup_evaluation considers every stone on the board, which implies that if a group contains more than one stone, then it will be counted more than once. This slightly inefficient method could be improved by ensuring that the markers (set up during PROCsearch to indicate that a stone has been counted) are not erased at the end of PROCcount. Note that the liberty markers should be erased at the end of each search, as different groups could share the same liberties. You must also make sure that the markers are only left when the count routine is being called from PROCgroup_evaluation. The position of all the liberties and associated counts would have to be saved, and the legality of each move checked later, when the markers have been cleared.

In the next instalment, we will add a 'catch' evaluation routine to provide a reasonable move in the absence of the need to attack or defend a group. The computer will then be able to give you a better game.

STACK 'EM UP

One of the major obstacles of FORTH programming is its syntax. In this instalment, we discuss the inherent logic behind the use of 'reverse Polish notation' and demonstrate how, in fact, its use simplifies arithmetical operations and the manipulation of data.

It is common sense that you can't get the sum of two numbers until you know what both the numbers are. This suggests that, although we write 2+3 with the + in the middle, a computer would rather get the two numbers before it starts to think about +. It really thinks in terms of 2 3 + (that is, get the two numbers and then add them.)

We're used to writing arithmetic operators like +, –, * and / in the middle ('infix notation') as in 2+2. It often reflects English ('two plus two makes four') and it clearly separates the operands from each other. These are good reasons for using infix notation in a computer language.

However, infix notation also has disadvantages that show up when you want your language to be extendable. First, an operator written this way must have exactly two operands (arguments or parameters), one on either side. For any more you need a functional notation, like FNf(a, b, c, d, e). Secondly, the notation is inherently ambiguous when you write something like 2+3*5. Which gets done first — the + or the *? It is only through a collection of extra rules, such as the one saying that * has a 'higher priority' than +, that this can be determined. If you want to extend the infix notation to new operators, you have to be able to extend the rules as well.

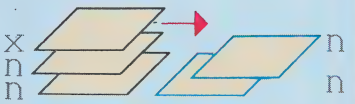
FORTH goes for the simplest possible solution, the one that the computer really wanted anyway. For *all* operators or functions (all words, in fact), whether traditional ones like + or new ones defined by you, you first write the operands and then the word itself — like 2 3 +. You can think of this as 'recipe book notation' — gather together the ingredients and then cook them. Its technical name is 'reverse' Polish notation (see page 1340). Forward Polish notation puts the operator *before* the operands and is what LOGO uses for new functions. At this point, we should point out that all the numbers handled by FORTH are integers. This is for reasons of efficiency, but it does mean that division does not normally give the exact fractional answer.

Suppose now you had a more complicated infix expression, like 2*3+8/4. According to the usual governing priority, the *last* operator to be done will be the +, so the final result will be the sum of 2*3

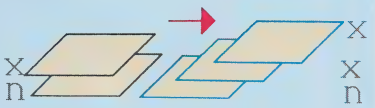
Stack Control

It isn't always possible to arrange the stack in such a way as to order numbers correctly for operations without first manipulating their positions in some way or another. FORTH provides a number of words for stack manipulation. Here are a few examples:

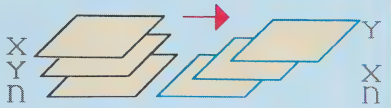
- DROP x – –
  (Throws away the top of the stack)



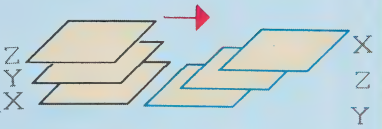
- DUP x – – x, x
  (Duplicates the top of the stack)



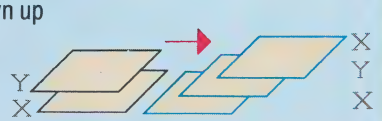
- SWAP x, y – – y, x
  (Swaps the top two elements on the stack)



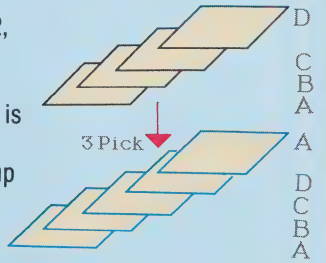
- ROT x, y, z – – y, z, x
  (Rotates the top three elements on the stack, so that the third number down is brought to the top)



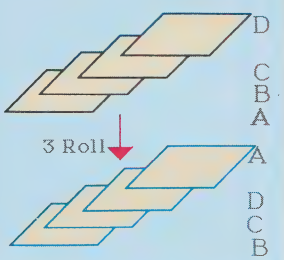
- OVER x, y – – x, y, x
  (Copies the second number down up over the top)



- PICK xn, . . ., x2, x1, x0, n – – xn, . . ., x2, x1, x0, xn
  (Like OVER, but you have to supply an extra number (n) to say which number is copied up over the top. For instance, 1 PICK is just like OVER, 2 PICK copies up the third number down, and 0 PICK is just like DUP)



- ROLL xn, . . ., x2, x1, x0, n – – . . ., x2, x1, x0, xn
  (This is like ROT, but with an extra number, as in PICK. 2 ROLL is just like ROT, and 3 ROLL rotates four numbers. 1 ROLL is just like SWAP)



FigFORTH lacks PICK and ROLL. FORTH-79 has them, but the extra number you supply is one more than in FORTH-83. Therefore, in FORTH-79, 3 ROLL is like ROT

and 8/4. Therefore, a first stage in writing this in reverse Polish notation is:

```
(2*3) (8/4) +
```

But this isn't right yet, because 2*3 and 8/4 are still in infix notation. Let's rewrite these too:

```
(2 3 *) (8 4 /) +
```

You might think you need the brackets to show how this is grouped, but in fact reverse Polish *never* needs brackets. The grouping is always unambiguous even without them. Indeed, FORTH uses brackets for something quite different — comments — so you are obliged to take the brackets out. This leaves the final reverse Polish rewriting of 2*3+8/4 as:

```
2 3 * 8 4 / +
```

Remember that in FORTH the spaces are vital.

We can now see how reverse Polish overcomes the two problems of extendability for infix notation. First, there is no reason why an operator should be restricted to two operands. It is just as easy to write one, three or four operands followed by the operator. The FORTH */, for example, has three operands: it multiplies the first two together, and divides the result by the third. This fits naturally into the reverse Polish system.

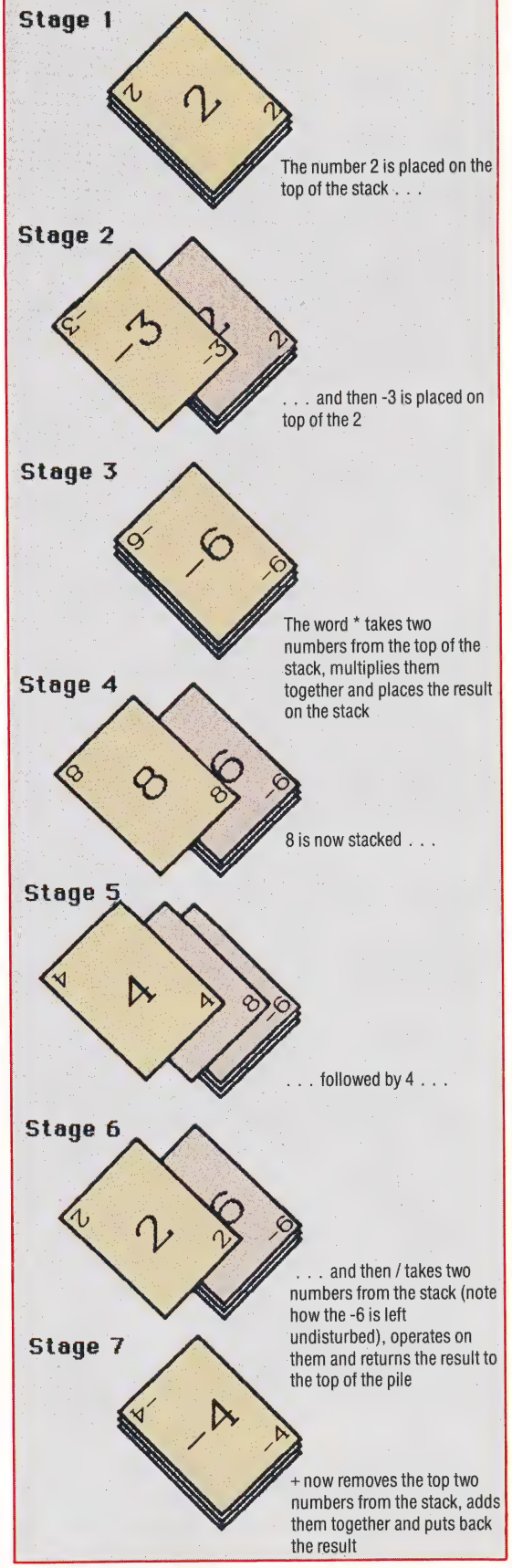
The second problem we found with infix notation was that it needed priority rules and brackets to make it unambiguous. Reverse Polish simply doesn't suffer from this. It tells the computer exactly how to calculate the result in an unambiguous way with no extra rules or brackets.

Let us now look at what the computer makes of a reverse Polish expression. The simplest illustration is with something like 2 3 +. FORTH encounters 2, commits that to memory, then encounters 3 and 'remembers' that. When it encounters + it knows — or, rather, it has to assume — that it has already remembered two numbers. It finds them, adds them together and 'remembers' the result in case there is either another calculation to come, or the result is to be used for some other purpose — printing to the screen, for example.

We presented a brief explanation of the way FORTH remembers numbers by 'pushing' them onto a stack — on a 'last in first out' (LIFO) basis — in the last instalment (see page 1446). This concept should be a familiar one with machine code programmers, but for those who find the stack concept confusing, here is a further example. To see how FORTH 'remembers' a number, imagine doing it by hand using a stack of cards on the table. To remember a number, you write it on a fresh card and put it on top of the stack. For +, you remove the top two cards and use the numbers from them, but you don't disturb the rest of the stack. For 2 -3 * 8 4 / + (for (2*-3)+(8/4)) the stack evolves as shown in the diagram. Notice how, at step 6 in the procedure, the card with -6 written on it is left undisturbed while you divide 8 by 4.

We should note at this point that variables are treated rather differently from the integers used in

Getting On Top Of It
This further example of stack operation shows how the different numbers are 'pushed onto' and 'pulled from' the stack execution of 2 -3 * 8 4 / + (2*-3+8/4 using infix notation)


Stage 1
The number 2 is placed on the top of the stack . . .
Stage 2
. . . and then -3 is placed on top of the 2
Stage 3
The word * takes two numbers from the top of the stack, multiplies them together and places the result on the stack
Stage 4
8 is now stacked . . .
Stage 5
. . . followed by 4 . . .
Stage 6
. . . and then / takes two numbers from the stack (note how the -6 is left undisturbed), operates on them and returns the result to the top of the pile
Stage 7
+ now removes the top two numbers from the stack, adds them together and puts back the result

CAROLINE CLAYTON

this example. A variable leaves its address on the stack. This can then be converted to the value of the variable (using @, the 'fetch' instruction) or it can be used to update the variable by means of the ! ('store') command.

Our arithmetic stack example clearly demonstrates the appropriateness of reverse Polish notation. FORTH maintains its own internal imitation of the stack of cards (called a 'data stack') and then for each number or word it encounters, the computer can do something definite without worrying about what went before or what is to come.

Here is another example of how to use the stack. The word 2* replaces the top of the stack, whatever it is, by itself doubled. It is defined thus:

```
: 2* ( n -- n*2)
   2 *
;
```

Note the use of brackets to provide comments, and the -- symbol, both of which are explained in the Taking Notes box. As an example of using our new word, 2*, the input:

```
19 2* .
```

displays 38 as its result. Note that the symbol . is the FORTH word for PRINT. It removes the top of the stack and displays it on the screen. You can see how the * in the definition of 2* expects at least two numbers on the stack. The second is the 2 included in the definition, but the first (19 in our example) is expected to be on the stack when 2* is used.

One advantage of stack-based operations is that an operation is allowed to produce more than one result. For example, the word /MOD leaves *two* numbers on the stack: the 'quotient' (answer) and the 'remainder' after a division. We could express this operation in another form, using the -- notation as follows:

```
m, n -- remainder, quotient of m/n
```

This would be impossible with infix notation, but with a stack it is quite natural.

Taking direct responsibility for the stack is a powerful feature of FORTH, but it can also present difficulties. For example, sometimes the numbers as supplied on the stack are not in quite the right order and you need some stack manipulations to rearrange them. Our Stack Control box lists some of the standard words used for this, and an example follows of a case where such manipulation might be needed. The word */ defined below is not quite as good as the one supplied as standard (which takes care to give the right answer even if the multiplication gives a result too big to fit on the stack), but it shows the manipulating words ROT and SWAP in action:

```
: */ (x,y,z -- x*y/z)
   ROT ROT *
   SWAP /
;
```

Here is how it works for the example 4 3 6 */:

Operation In */	Stack (Top →)
At the start of */	4, 3, 6
ROT	3, 6, 4
ROT	6, 4, 3
*	6, 12
SWAP	12,6
/	2
	Empty (and 2 displayed)

Fortunately, the minute details of stack manipulation can usually be kept under the carpet by hiding them inside word definitions.

Taking Notes

You must take great care that you put exactly as many numbers on the stack as a word needs. If you put on too few, it will use some from further down that were supposed to be left undisturbed, and if you put on too many, the extra ones will get in the way.

To say very clearly what a word needs from the stack, and what it leaves behind, you can use the '--' notation. Write a list of what the word removes, then '--', and then a list of what it leaves behind. Here are some examples:

Word:	Effect:
+	m, n -- m+n
.	m --
*/	x, y, z -- x*y/z

For each list, the top of the stack is always stated last. The '--' notation is not a part of the FORTH language, but a feature added to allow us to understand a listing better. You will find it very useful to put it in FORTH word definitions in comments (enclosed in brackets). You'll need to remember that the first bracket needs a space after it — because it is a word that means 'ignore everthing up to the closing bracket'

Displaying The Stack

A word you'll find very useful is .S, which displays the stack. It's not in standard FORTH, but in FORTH-83 you can define it by:

```
: .S (--)
   ( displays the stack, from the bottom up )
   0 DEPTH 1 - DO
      I PICK .
   -1 +LOOP
;
```

In FORTH-79, because PICK works differently, you must replace the third line by:

```
1 DEPTH DO
```

DEPTH (-- stack depth) tells you how many numbers there were on the stack before you did DEPTH. However, figFORTH has neither DEPTH nor PICK, so this definition won't work on that dialect

WHEN IN ROM

The Interface 1 can be used to allow you to add your own BASIC commands to the Spectrum BASIC. This facility was an essential component in the design of the device because it enabled the use of certain commands to control the Microdrives and so on. Here, we examine the 'theory' behind the addition of new commands.

New BASIC commands are added by 'deceiving' the error-handling routines of the Spectrum into accepting something that they normally wouldn't. So, a good place to start our discussion is to look at the normal error-trapping techniques of the Sinclair Spectrum.

When a line of BASIC is typed in, and ENTER is pressed, the line is immediately examined by the BASIC interpreter to see if it is syntactically correct. Errors such as missing variable names after NEXT commands, incorrect use of parameters, and so on, are all detected at this stage. This is why it's not possible to enter a 'dud' line of BASIC into the Spectrum. If it's correct, then it is fitted into the appropriate place in the program if it is a program line, or executed if it is a direct command.

If the line isn't correct, however, then an RST #8 instruction is executed, followed by a DEFB, which indicates to the operating system the nature of the error that has occurred. This leads to the display of the flashing ? — which no doubt you have already encountered.

Once a statement is executed, either as a direct command or as a program line, then the syntax of the line is checked again. During this second check, for example, any numeric expressions in the line are evaluated and the line is then executed by calling the relevant ROM routines. Errors such as No Such Variable are detected at this point.

The first check, known as the *syntax check*, doesn't execute the command, but simply ensures that it is syntactically valid. The second check — the *run-time check* — actually executes the statement. However, when the Interface 1 (IF1) hardware is added to the system, this process is modified. As we have already seen in this series, the hardware of the Interface 1 ensures that whenever address #08 or #1708 is accessed, the shadow ROM is paged in and the main ROM is switched out. The Spectrum system is now said to be operating in the 'shadow ROM environment', and this situation continues until address #700 of the IF1 ROM is accessed — at which point the shadow ROM is disabled and the main ROM takes charge again.

From this description, it is clear that when an error occurs the shadow ROM is immediately paged in, and a second examination of the condition that caused the error is carried out. If the error was generated by a command such as CAT or FORMAT, which are legal with the IF1 fitted, then it is dealt with by the IF1 ROM and a return made to the main ROM.

If the error condition did not arise from one of these commands, however, then a jump is made to the address held in the IF1 ROM system variable called VECTOR (which is held at addresses 23735 and 23736). VECTOR holds the address of the shadow ROM error handler. It is important to note that this address will differ in different versions of the Interface 1 ROM. We have used the first version of the IF1 ROM. In the next instalment, however, we will discuss some of the main differences between the shadow ROM versions, and provide you with a method of identifying which release version you are using.

For the first version of the IF1 ROM, VECTOR holds #1F0, so in error situations a call to this routine will eventually lead to a return to the editor at 'syntax checking' time in order to edit the offending item in the BASIC line that is being checked. Thus, if we alter the address held in VECTOR, we can redirect the Spectrum error handler so that it executes a routine of our own devising instead of the standard error routine. We will see how this is done in the next instalment. Before we do that, let's look at some shadow ROM routines that will be of use to us in this process.

#0010

This routine allows you to call a main ROM routine when in the shadow ROM environment. It is the only means we have of calling a main ROM routine, because we have to take care of the paging in and out of the main ROM. It is used in the following way:

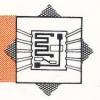
```
RST     #0010
DEFW    address     ; address of routine to be called
```

#0020

This routine is the shadow ROM version of RST #8 in the main ROM. RST #0020, followed by a single byte which indicates the error message to be generated, will do the job. The following table shows the range of error messages:

Out Of The Shadows

In addition to the Microdrive, RS232 and networking facilities provided by Interface 1, the shadow ROM also has a number of useful routines, including a decimal-to-hex converter (which does not appear to be taken up by any of the other system subroutines). To make best use of the interface from machine code, we recommend Melbourne House's book Spectrum Shadow ROM Disassembly (ISBN 0-86161-191-8), which gives full details of the shadow ROM routines, as well as some useful utility listings



Byte	Error Message
00	Nonsense in BASIC
01	Invalid Stream Number
02	Invalid Device Expression
03	Invalid Name
04	Invalid Drive Number
05	Invalid Station Number
06	Missing Name
07	Missing Station Number
08	Missing Drive Number
09	Missing Baud Rate
10	Header Mismatch Error
11	Stream Already Open
12	Writing to a Read File
13	Reading a Write File
14	Drive Write Protected
15	Microdrive Full
16	Microdrive Not Present
17	File Not Found
18	Hook Code Error
19	CODE Error
20	MERGE Error
21	Verification Has Failed
22	Wrong File Type
255	Program Finished

#0028

This allows us to generate a 'normal' error message — that is, one that is generated by the main ROM and not the shadow ROM. Before this routine is called, address 23610 is loaded with the relevant error code.

#01F0

This generates the 'syntax error' flashing cursor while syntax checking is being performed.

#05B7

This is the routine that is called to indicate that the syntax of the command has been checked. It also checks for the end of a BASIC statement. We'll consider its use in detail later.

#05C1

This is used to conclude command execution in the shadow ROM, and passes control back to the main ROM.

#0700

This forces a return to the main ROM environment. It can be seen as a means of paging *out* the shadow ROM and paging the main ROM *in*. Finally, let's consider the relevant hook code:

- *Hook Code 50:* Like all hook codes, this should *not* be used in the shadow ROM environment. It enables you to call shadow ROM routines from within the main ROM, and should only be used if you have a good knowledge of the addresses within the shadow ROM. It is entered with the address of the shadow ROM routine in the HL register routine, as the following code shows:

```
LD      HL, address ; address of routine
LD      (23789), HL
RST     #8
DEFB    50
```

There are other points of interest about the shadow ROM environment. It should be clear from our discussion that the Z80 'restart' commands are all different. The keyboard is not scanned when in the shadow ROM, and the FRAMES system variable is not incremented. Thus, any length of time spent in the shadow ROM will tend to slow down the FRAMES variable and make it a little unreliable for timing applications.

One problem that is occasionally encountered is in the use of the 'floating point calculator' (FPC) from the shadow ROM. The routine at #0010 in the shadow ROM seems unable to cope with calling the FPC, so it's best to exit the shadow ROM and work in the main ROM to use it. If you then want to re-enter the shadow ROM, you can do so using hook code 50.

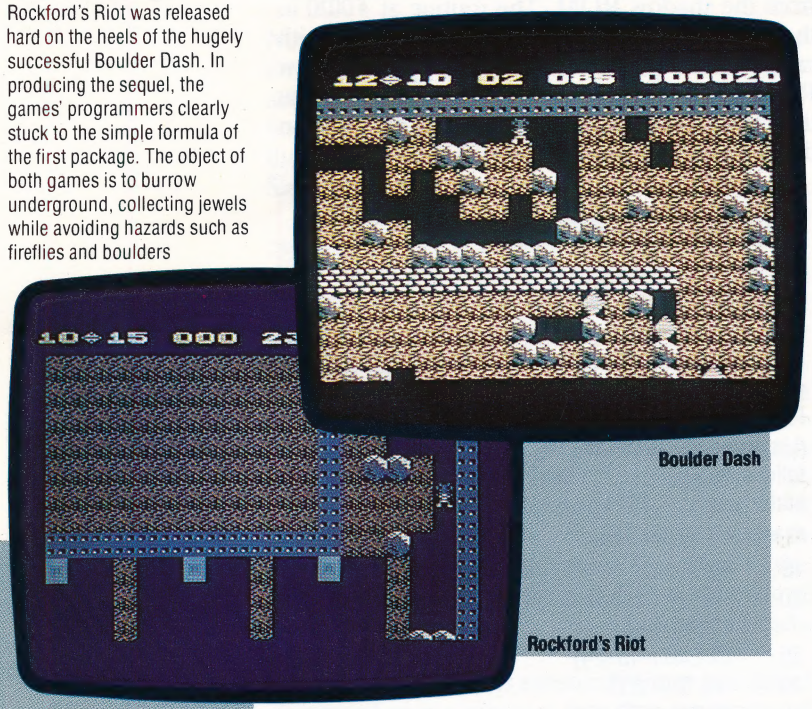
Taking A PEEK

You might like to examine some of the shadow ROM routines mentioned here yourself. Disassembling other people's programs can be a useful learning experience and can also help you to use the routines provided more efficiently. Unfortunately, you cannot PEEK the shadow ROM directly from BASIC because it is only paged in during Interface 1 operations. The following short BASIC program will load any portion of the shadow ROM into RAM, where you can examine it at your leisure

```
10    GO SUB 200
20    INPUT "How many bytes do you want to
      CLEAR?";b
30    CLEAR r-(b+24)
40    GO SUB 200
50    FOR n=1 TO 23
60    READ d: POKE r+n,d
70    NEXT n
80    INPUT "Start address in Shadow ROM?";s
90    INPUT "Number of bytes to copy?";b
100   LET z=s: GO SUB 300
110   POKE r+12,l: POKE r+13,h
120   LET z=b: GO SUB 300
130   POKE r+18,l: POKE r+19,h
140   LET z=r+9: GO SUB 300
150   POKE r+2,l: POKE r+3,h
160   LET z=r+24: GO SUB 300
170   POKE r+15,l: POKE r+16,h
180   RANDOMIZE USR (r+1)
190   PRINT "Data stored at" :r+24
195   STOP
200   LET r=PEEK 23730+256*PEEK 23731:
      RETURN
300   LET h=INT (z/256): LET l=z-256*h: RETURN
400   DATA 33,0,0,34,237,92,207,50,
      225,225,33,0,0,17,0,0,1,0,0,237,176,
      199,201,
```

THE ROCKFORD FILES

As with the motion picture and publishing industries, software houses have realised that sequels to popular products are sure bets to generate initial sales. We play two games from the US, Boulder Dash and its follow-up, Rockford's Riot, both of which are now available in the UK.



There's A Riot Going On
Rockford's Riot was released hard on the heels of the hugely successful Boulder Dash. In producing the sequel, the games' programmers clearly stuck to the simple formula of the first package. The object of both games is to burrow underground, collecting jewels while avoiding hazards such as fireflies and boulders

Although the software games market has developed into a multi-million pound industry, with literally thousands of different games being produced each year, few make lasting impressions. Quite often, even the most successful game is forgotten within a few months of its release. There are, however, a handful of games that produce dedicated followings. In the UK, for example, two such games are Manic Miner and its follow-up Jet Set Willy. The latter has been so successful that its publisher, Software Projects, has bowed to public pressure and re-issued the game with extra rooms added.

In the US, an avid following has built up around Rockford, the hero of Boulder Dash — which is also a game about 'mining' — and its success has promoted a sequel, called Rockford's Riot (in the UK) or Boulder Dash II (in the US). The idea behind both games is to collect 'jewels', which are buried underground; Rockford must burrow through the earth in order to reach them.

Of course, a computer game would be incomplete without obstacles in the way. The primary problem is the boulders scattered around the screen, which prevent Rockford from digging straight to the jewels. If a boulder falls on Rockford, the player will lose a life, even though it is possible for our hero to stand with a rock resting on his head. Although it would appear that the boulders are a nuisance, they do have their uses.

Another of the menaces are the fireflies, creatures that cannot dig tunnels themselves, but can move through the existing tunnels, or those created by Rockford. They are deadly if they come within a square of Rockford, but fortunately they have their weaknesses. First, when given a choice of paths to enter, they always choose the left-hand route. Secondly, they are as vulnerable to falling rocks as Rockford is. Therefore, you can predict where the fireflies are going, wait at the end of a tunnel and drop a rock on them when they appear.

This facet of the game provides the solution to another problem. Often, the jewels will be behind a wall or some other obstacle preventing Rockford's passage. By dropping a rock on a firefly that is next to a wall, you can blast both the creature and a hole through the barrier.

To get to the next level, a certain number of jewels have to be collected, which will activate a door somewhere on the screen through which Rockford can pass. Players new to the game may find this confusing at first because it appears that there are not enough jewels scattered around to collect. For example, both games contain an 'amoeba', a green mass that slowly expands to fill the screen. However, if blocked off by boulders so that it cannot expand, the amoeba will reach a 'critical mass' and crystallise into jewels. In other screens, jewels are created by dropping boulders on particular objects.

Much of the action depends on fast reflexes with the keyboard or joystick, but the mark of a good game is that it will require well thought out moves as well. In this respect, both games score heavily. Boulder Dash and Rockford's Riot depend, to a very large extent, on the player's cunning and strategy. Often, a precise and complex sequence of actions must be performed in order to achieve the required result.

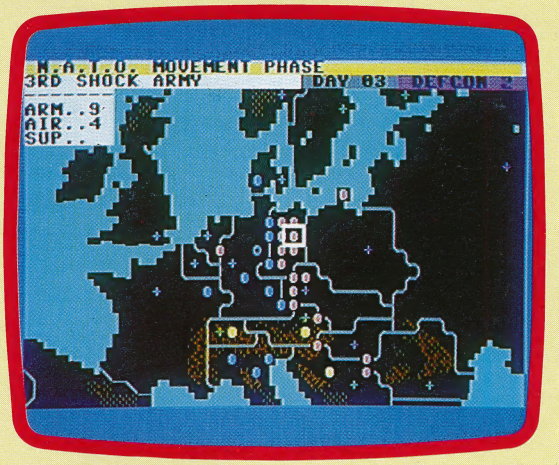
The graphics in both games, which are very similar, are excellent. The screens are created in multicolour mode and the scrolling and user-control are both impeccable. Another feature of the games that makes them such a delight to play is the attention to detail that has been put into the design of the screen displays.

Boulder Dash and Rockford's Riot: For both the Commodore 64 and Sinclair Spectrum
Price: £9.95
Publishers: Beyond Software, Competition House, Farndon Road, Market Harborough, LE16 9NR
Authors: Peter Liepa with Chris Gray
Format: Cassette
Joysticks: Not required

THE UNTHINKABLE

Our occasional series of reviews of chart-topping games for the more popular home micros continues with a look at two packages for the Commodore 64. Bobby Pickering tries his hand at Raid Over Moscow and Theatre Europe, fantasy games that will strike fear and loathing into the hearts of many campaigners for nuclear disarmament

Raid Over Moscow, from US Gold, is a conventional 'shoot-em-up' and score scenario, while PSS's Theatre Europe is based on a traditional strategy game concept. The popularity of these two packages demonstrates how many of the chart-topping games for home micros are not necessarily the most innovative, but rather games written to well-tried formulas. It is a well-known marketing principle — in pop music as well as pop software — that products do well if they are a little outrageous. So here, traditional computer game concepts are packaged in a scenario that makes reference to rather unpalatable realities. As we accustom ourselves daily to the madness of a possible nuclear exchange, these games allow us to fantasise the unthinkable. For about £10 you can blast away at the Soviet Union or press the button to start an all-out nuclear war.



Raid Over Moscow (cassette: £9.95; disk: £12.95) begins with a space-eye view of the northern hemisphere, showing the US and USSR picked out in yellow and red respectively (surely no colour symbolism is intended?) High above the planet is an orbiting American mothership. At the bottom of the screen a message confirms that an enemy missile has been detected, stating which city in the USSR it was launched from and which US city it is destined for. Your mission (should you decide to accept it) is to board a fighter ship inside the mothership, negotiate an exit from the space station, and then blast your way across some bizarre 'Russian' countryside (picking off a bus full of innocent peasants if that takes you fancy), until you reach your destination. Once inside the Soviet city of your computer's choice, you are presented with a screen vaguely reminiscent of Red Square (all Russian metropoli, of course, look exactly the same). Your task is to pick off snipers on the surrounding rooftops, blast away the towers and doors of the palace at the other end of the square, and enter the reactor room at the heart of the palace.

This room looks exactly like a glorified squash court in which a small robot (the device controlling the missile heading stateside) trundles back and forth. You must bounce a disc off the back wall so that it destroys the robot. And your mission is complete.

Now if this scenario sounds a little too far-fetched or bloodthirsty, you have an alternative. Theatre Europe (cassette:£9.95) focuses on the countries of Europe as the battleground for nuclear Armageddon. The game owes a lot to traditional strategic boardgames, such as Diplomacy. In fact, its primary screen display is a map of Europe on which two opposing armies — those of NATO and the Warsaw Pact — are represented. By careful management of your forces (you can play either side) you must weather 30 days of military build-up that can lead to an all-out nuclear strike.

As in the real world, you can decide at any stage to indulge in a little tactical 'limited engagement' battle. But there's few thrills here — an unimaginative screen depicting rather bleak coutryside is displayed, and you have to blast away at the tanks and planes that advance across it. No wonder then, that the option to indulge in an all-out nuclear strike becomes so tempting. Once you've worked out the password to set your missiles flying, and hummed along to a few bars of John Lennon's Give Peace A Chance, you can watch the remarkable final engagement. After the missiles from both sides traverse the map of Europe (like swarms of bees on a radar screen), a city skyline flashes up, and is soon overshadowed by a mushroom cloud.

At the end of the day, when all is bombed and done, anyone playing these games may be brought a little closer to an understanding of Europe's place sandwiched between two superpowers. And the possible consequences of our precarious position. . .

© 1983 WEIL, STRAUSS, BARZEL—BROWN UNIV.